The Python Library Reference

출시 버전 *3.7.17*

Guido van Rossum
and the Python development team

6월 28, 2023

Contents

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, 수천 가지 컴포넌트(개별 프로그램과 모듈부터 패키지 및 전체 응용 프로그램 개발 프레임워크까지)가 늘어나고 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 X에서 지원됩니다.

CHAPTER 2

내장 함수

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

		내장 함수		
abs()	*delattr()*	*hash()*	*memoryview()*	*set()*
all()	*dict()*	*help()*	*min()*	*setattr()*
any()	*dir()*	*hex()*	*next()*	*slice()*
ascii()	*divmod()*	*id()*	*object()*	*sorted()*
bin()	*enumerate()*	*input()*	*oct()*	*staticmethod()*
bool()	*eval()*	*int()*	*open()*	*str()*
breakpoint()	*exec()*	*isinstance()*	*ord()*	*sum()*
bytearray()	*filter()*	*issubclass()*	*pow()*	*super()*
bytes()	*float()*	*iter()*	*print()*	*tuple()*
callable()	*format()*	*len()*	*property()*	*type()*
chr()	*frozenset()*	*list()*	*range()*	*vars()*
classmethod()	*getattr()*	*locals()*	*repr()*	*zip()*
compile()	*globals()*	*map()*	*reversed()*	*__import__()*
complex()	*hasattr()*	*max()*	*round()*	

abs(*x*)
숫자의 절댓값을 돌려줍니다. 인자는 정수 또는 실수입니다. 인자가 복소수면 그 크기가 반환됩니다.

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
```

any(*iterable*)

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

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

ascii(*object*)

repr() 처럼, 객체의 인쇄 가능한 표현을 포함하는 문자열을 반환하지만, `\x` 나 `\u` 또는 `\U` 이스케이프를 사용하여 *repr()* 이 돌려주는 문자열에 포함된 비 ASCII 문자를 이스케이프 합니다. 이것은 파이썬 2의 *repr()* 이 돌려주는 것과 비슷한 문자열을 만듭니다.

bin(*x*)

정수를 "0b" 가 앞에 붙은 이진 문자열로 변환합니다. 결과는 올바른 파이썬 표현식입니다. *x* 가 파이썬 *int* 객체가 아니라면, 정수를 돌려주는 `__index__()` 메서드를 정의해야 합니다. 몇 가지 예를 들면:

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

접두어 "0b" 가 필요할 수도, 필요 없을 수도 있다면, 다음 방법의 하나를 사용할 수 있습니다.

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

자세한 내용은 *format()*을 보세요.

class bool([*x*])

논리값, 즉 `True` 또는 `False` 중 하나를 돌려줍니다. *x* 표준 논리값 검사 절차 를 사용하여 변환됩니다. *x* 가 거짓이거나 생략되면 `False` 를 돌려줍니다. 그렇지 않으면"True" 를 돌려줍니다. *bool* 클래스는 *int* (숫자 형 — *int, float, complex* 참조)의 서브 클래스입니다. 서브 클래스를 더 만들 수 없습니다. 이것의 유일한 인스턴스는 `False` 와"True" 입니다 (논리값를 보세요).

버전 3.7에서 변경: *x*는 이제 위치 전용 매개 변수입니다.

breakpoint(**args*, ***kws*)

이 함수는 호출 지점에서 디버거로 진입하게 만듭니다. 특히 *sys.breakpointhook()* 을 호출하고 `args` 와 `kws` 를 그대로 전달합니다. 기본적으로, `sys.breakpointhook()` 은 인자를 기대하지 않고 *pdb.set_trace()*를 호출합니다. 이 경우, 이것은 순전히 편의 기능이므로 *pdb*를 명시적으로 임포트하거나 디버거에 들어가기 위해 많은 코드를 입력할 필요가 없습니다. 그러나, *sys.breakpointhook()* 은 다른 함수로 설정될 수 있고, *breakpoint()* 는 그것을 자동으로 호출하여, 선택한 디버거에 들어갈 수 있도록 합니다.

버전 3.7에 추가.

class bytearray([*source*[, *encoding*[, *errors*]]])

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 *(buffer)* 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*[, *encoding*[, *errors*]]])

새로운 "바이트열" 객체를 돌려줍니다. 이 객체는 `0 <= x < 256` 범위에 있는 정수의 불변 시퀀스입니다. *bytes* 는 *bytearray* 의 불변 버전입니다 – 같은 불변 메서드와 같은 인덱싱 및 슬라이싱 동작을 갖습니다.

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.

버전 3.2에 추가: 이 함수는 파이썬 3.0에서 먼저 제거된 다음 파이썬 3.2에서 다시 도입했습니다.

chr(*i*)

유니코드 코드 포인트가 정수 *i* 인 문자를 나타내는 문자열을 돌려줍니다. 예를 들어, `chr(97)` 은 문자열 `'a'` 를 돌려주고, `chr(8364)` 는 문자열 `'€'` 를 돌려줍니다. 이 것은 *ord()* 의 반대입니다.

인자의 유효 범위는 0에서 1,114,111(16진수로 0x10FFFF) 까지입니다. *i* 가 이 범위 밖에 있을 때 *ValueError* 가 발생합니다.

@classmethod

메서드를 클래스 메서드로 변환합니다.

인스턴스 메서드가 인스턴스를 받는 것처럼, 클래스 메서드는 클래스를 묵시적인 첫 번째 인자로 받습니다. 클래스 메서드를 선언하려면 이 관용구를 사용합니다:

```
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, *staticmethod()*를 보세요.

클래스 메서드에 대한 더 자세한 정보를 원하면, types을 참고하세요.

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 퓨처 문이 *source* 의 컴파일에 영향을 미칠지 제어합니다. 둘 다 제공되지 않는 경우 (또는 둘 다 0의 경우), 코드는 *compile()* 을 호출하는 코드에 적용되고 있는 퓨처 문으로 컴파일됩니다. *flags* 인자가 주어지고, *dont_inherit* 가 없으면 (또는 0) 원래 사용될 것에 더해 *flags* 인자로 지정된 퓨처 문이 사용됩니다. *dont_inherit* 가 0이 아닌 정수면 *flags* 인자가 사용됩니다 – compile 을 호출하는 코드에 적용되는 퓨처 문은 무시됩니다.

퓨처 문은 여러 개의 문장을 지정하기 위해 비트 OR 될 수 있는 비트에 의해 지정됩니다. 주어진 기능을 지정하는 데 필요한 비트 필드는 *__future__* 모듈의 `_Feature` 인스턴스에서 `compiler_flag` 어트리뷰트로 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` (assert가 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.

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: 윈도우 및 맥의 줄 바꿈을 사용할 수 있습니다. 또한, 이제는 `'exec'` mode에서 입력이 줄 넘김 문자로 끝나지 않아도 됩니다. *optimize* 매개변수가 추가되었습니다.

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

class complex([*real*[, *imag*]])

real + *imag**1j 값을 가진 복소수를 돌려주거나 문자열 또는 숫자를 복소수로 변환합니다. 첫 번째 매개변수가 문자열이면 복소수로 해석되며, 두 번째 매개변수 없이 함수를 호출해야 합니다. 두 번째 매개변수는 결코 문자열 일 수 없습니다. 각 인자는 모든 (복소수를 포함한) 숫자 형이 될 수 있습니다. *imag* 가 생략되면 기본값은 0이고, 생성자는 *int* 와 *float*와 같은 숫자 변환으로 사용됩니다. 두 인자가 모두 생략되면 `0j` 를 돌려줍니다.

참고: 문자열을 변환할 때, 문자열은 중앙의 + 또는 - 연산자 주위에 공백을 포함해서는 안 됩니다. 예를 들어, `complex('1+2j')` 는 괜찮지만 `complex('1 + 2j')` 는 *ValueError* 를 일으킵니다.

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

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

delattr(*object*, *name*)

이것은 *setattr()* 의 친척뻘입니다. 인자는 객체와 문자열입니다. 문자열은 객체의 어트리뷰트 중

하나의 이름이어야 합니다. 이 함수는 객체가 허용하는 경우 명명된 어트리뷰트를 삭제합니다. 예를 들어, `delattr(x, 'foobar')` 는 `del x.foobar` 와 동등합니다.

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)
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([*object*])
인자가 없으면, 현재 지역 스코프에 있는 이름들의 리스트를 돌려줍니다. 인자가 있으면, 해당 객체에 유효한 어트리뷰트들의 리스트를 돌려주려고 시도합니다.

객체에 `__dir__()` 메서드가 있으면, 이 메서드가 호출되는데, 반드시 어트리뷰트 리스트를 돌려줘야 합니다. 이렇게 하면 커스텀 `__getattr__()` 또는 `__getattribute__()` 함수를 구현하는 객체가 *dir()* 이 어트리뷰트들을 보고하는 방법을 커스터마이즈할 수 있습니다.

객체가 `__dir__()` 을 제공하지 않으면, 함수는 (정의되었다면) 객체의 `__dict__` 어트리뷰트와 형 객체로부터 정보를 수집하기 위해 최선을 다합니다. 결과로 얻어지는 리스트는 반드시 완전하지는 않으며, 객체가 커스텀 `__getattr__()` 을 가질 때 부정확할 수도 있습니다.

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']
>>> 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*)
두 개의 (복소수가 아닌) 숫자를 인자로 취하고 정수 나누기를 사용할 때의 몫과 나머지로 구성된 한 쌍의 숫자를 돌려줍니다. 두 인자의 형이 다른 경우, 이 항 산술 연산자에 대한 규칙이 적용됩니다. 정수의 경우, 결과는 `(a // b, a % b)` 와 같습니다. 부동 소수점 숫자의 경우 결과는 `(q, a % b)` 인데, *q*

는 보통 `math.floor(a / b)` 이지만, 이보다 1 작을 수 있습니다. 어떤 경우건 `q * b + a % b` 는 *a* 에 매우 가깝습니다. `a % b` 는 0 이 아닐 때 *b* 와 같은 부호를 가지며, `0 <= abs(a % b) < abs(b)` 가 성립합니다.

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 0 을 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(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

eval(*expression*[, *globals*[, *locals*]])

인자는 문자열 및 선택적 globals 및 locals 다. 제공된 경우, *globals* 는 딕셔너리여야 합니다. 제공되는 경우, *locals* 는 모든 매핑 객체가 될 수 있습니다.

expression 인자는 전역 및 지역 이름 공간으로 *globals* 및 *locals* 딕셔너리를 사용하여 파이썬 표현식 (기술적으로 말하면, 조건 목록) 으로 파싱 되고 값이 구해집니다. *globals* 사전이 제공되고 키 '__builtins__' 의 값을 담고 있지 않으면, *expression* 를 구문 분석하기 전에 내장 모듈 *builtins* 의 딕셔너리에 대한 참조를 그 키로 삽입합니다. 이는 *expression* 이 일반적으로 표준 *builtins* 모듈에 대한 모든 액세스 권한을 가지며 제한된 환경이 전파됨을 뜻합니다. *locals* 딕셔너리를 생략하면 기본적으로 *globals* 딕셔너리가 사용됩니다. 두 딕셔너리가 모두 생략되면, 표현식은 *eval()* 이 호출되는 환경에서 실행됩니다. 반환 값은 계산된 표현식의 결과입니다. 문법 에러는 예외로 보고됩니다. 예:

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

이 함수는 임의의 코드 객체 (*compile()* 로 만든 것과 같은) 를 실행하는 데에도 사용할 수 있습니다. 이 경우 문자열 대신 코드 객체를 전달합니다. 코드 객체가 mode 인자 `'exec'` 로 컴파일되었다면, *eval()* 의 반환 값은 `None` 입니다.

힌트: 문장의 동적 실행은 *exec()* 함수에 의해 지원됩니다. *globals()* 와 *locals()* 함수는 각각 현재의 전역과 지역 딕셔너리를 반환하는데, *eval()* 또는 *exec()* 에 전달하는 데 유용합니다.

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

exec(*object*[, *globals*[, *locals*]])

이 함수는 파이썬 코드의 동적 실행을 지원합니다. *object* 는 문자열 또는 코드 객체여야 합니다. 문자열이면 문자열은 파이썬 문장들의 스위트로 파싱된 후 (문법 에러가 발생하지 않는 한) 실행됩니다.[1] 코드 객체면, 단순히 실행됩니다. 모든 경우에, 실행되는 코드는 파일 입력으로 올바를 것이 기대됩니다 (레퍼런스 설명서의 "파일 입력" 섹션을 보세요). `return` 과 `yield` 문은 *exec()* 함수에 전달 된 코드 문맥 안에서조차도 함수 정의 밖에서 사용될 수 없음에 유의하세요. 반환 값은 `None` 입니다.

모든 경우에, 선택적 부분을 생략하면, 현재 스코프에서 코드가 실행됩니다. *globals* 만 제공된 경우, 사전이어야 하며, 전역과 지역 변수 모두에 사용됩니다. *globals* 및 *locals* 가 주어지면, 전역과 지역 변수에 각각

[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.

사용됩니다. 제공되는 경우, *locals* 는 모든 매핑 객체가 될 수 있습니다. 모듈 수준에서, 전역과 지역은 같은 딕셔너리임을 기억하세요. exec 가 *globals* 와 *locals* 로 별도의 객체를 받으면, 코드는 클래스 정의에 포함된 것처럼 실행됩니다.

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.

참고: 내장 함수 *globals()* 와 *locals()* 는 각각 현재 전역 및 지역 딕셔너리를 돌려주는데, *exec()* 로 전달되는 두 번째 및 세 번째 인자로 사용하는 데 유용합니다.

참고: 기본 *locals* 는 아래 함수 *locals()* 에 설명된 대로 작동합니다: 기본 *locals* 사전에 대해 수정이 시도되어서는 안 됩니다. 함수 *exec()* 가 돌아온 후에 *locals* 에 코드가 만든 효과를 보려면 명시적으로 *locals* 딕셔너리를 전달해야 합니다.

---

filter(*function*, *iterable*)

function 이 참을 돌려주는 *iterable* 의 요소들로 이터레이터를 구축합니다. *iterable* 은 시퀀스, 이터레이션을 지원하는 컨테이너 또는 이터레이터 일 수 있습니다. *function* 이 `None` 이면, 항등함수가 가정됩니다, 즉, 거짓인 *iterable* 의 모든 요소가 제거됩니다.

`filter(function, iterable)` 는 function 이 `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.

function 이 거짓을 돌려주는 *iterable* 의 요소들을 돌려주는 상보적인 함수는 *itertools.filterfalse()* 를 보세요.

class float([*x*])

숫자 또는 문자열 *x* 로 부터 실수를 만들어 돌려줍니다.

인자가 문자열이면, 십진수를 포함해야 하고, 선택적으로 부호가 앞에 오며 선택적으로 공백으로 둘러싸일 수 있습니다. 선택적 부호는 `'+'` 또는 `'-'` 일 수 있습니다; `'+'` 부호는 생성되는 값에 아무런 영향을 주지 않습니다. 인자는 NaN (not-a-number) 또는 양 또는 음의 무한대를 나타내는 문자열 일 수도 있습니다. 더욱 정확하게, 입력은 앞과 뒤의 공백 문자를 제거한 후 다음 문법을 따라야 합니다:

```
sign            ::=   "+" | "-"
infinity        ::=   "Infinity" | "inf"
nan             ::=   "nan"
numeric_value   ::=   floatnumber | infinity | nan
numeric_string  ::=   [sign] numeric_value
```

여기서 `floatnumber` 는 floating 에 설명된 파이썬 실수 리터럴의 형식입니다. 대/소문자는 중요하지 않아서, 예를 들면, “inf”, “Inf”, “INFINITY” 및 “iNfINity”는 모두 양의 무한대에 대해 허용되는 철자입니다.

그렇지 않으면, 인자가 정수 또는 실수면 (파이썬의 부동 소수점 정밀도 내에서) 같은 값을 가진 실수가 반환됩니다. 인자가 파이썬 float 범위를 벗어나면, *OverflowError* 가 발생합니다.

일반적인 파이썬 객체 `x` 의 경우, `float(x)` 는 `x.__float__()` 로 위임합니다.

인자가 주어지지 않으면, `0.0` 을 돌려줍니다.

예:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

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: *x*는 이제 위치 전용 매개 변수입니다.

format(*value*[, *format_spec*])

format_spec 의 제어에 따라, *value* 를 "포맷된" 표현으로 변환합니다. *format_spec* 의 해석은 *value* 인자의 형에 의존하지만, 대부분의 내장형에 의해 사용되는 표준 포매팅 문법이 있습니다: 포맷 명세 미니 언어.

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

`format(value, format_spec)` 에 대한 호출은 `type(value).__format__(value, format_spec)` 로 번역되는데, value의 `__format__()` 메서드를 검색할 때 인스턴스 딕셔너리를 건너뜁니다. 메서드 검색이 *object* 에 도달하고 *format_spec* 이 비어 있지 않거나, *format_spec* 또는 반환 값이 문자열이 아닌 경우 *TypeError* 예외가 발생합니다.

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*[, *default*])

주어진 이름의 *object* 어트리뷰트를 돌려줍니다. *name* 은 문자열이어야 합니다. 문자열이 객체의 어트리뷰트 중 하나의 이름이면, 결과는 그 어트리뷰트의 값입니다. 예를 들어, `getattr(x, 'foobar')` 는 `x.foobar` 와 동등합니다. 명명된 어트리뷰트가 없으면, *default* 가 제공되는 경우 그 값이 반환되고, 그렇지 않으면 *AttributeError* 가 발생합니다.

globals()

현재 전역 심볼 테이블을 나타내는 딕셔너리를 돌려줍니다. 이것은 항상 현재 모듈의 딕셔너리입니다 (함수 또는 메서드 내에서, 이 모듈은 그것들을 호출하는 모듈이 아니라, 그것들이 정의된 모듈입니다).

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: 커스텀 `__hash__()` 메서드를 가진 객체의 경우, *hash()* 는 호스트 기계의 비트 폭을 기준으로 반환 값을 잘라 버리는 것에 주의하세요. 자세한 내용은 `__hash__()` 를 보세요.

---

help([*object*])
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.

*help()*를 호출할 때, 함수의 매개 변수 목록에 슬래시(/)가 표시되면, 슬래시 이전 매개 변수는 위치 전용이라는 것을 의미합니다. 자세한 내용은, 위치 전용 매개 변수에 대한 FAQ 항목을 참조하십시오.

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(*x*)
정수를 "0x" 접두사가 붙은 소문자 16진수 문자열로 변환합니다. *x* 가 파이썬 *int* 객체가 아니면, 정수를 돌려주는 `__index__()` 메서드를 정의해야 합니다. 몇 가지 예:

```
>>> 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 implementation detail: This is the address of the object in memory.

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.

class int([*x*])

class int(*x*, *base=10*)

숫자나 문자열 *x* 로 부터 만들어진 정수 객체를 돌려줍니다. 인자가 주어지지 않으면 `0` 을 돌려줍니다. *x* 가 `__int__()` 를 정의하면, `int(x)` 는 `x.__int__()` 를 돌려줍니다. *x* 가 `__trunc__()` 를 정의하면, `x.__trunc__()` 를 돌려줍니다. 실수의 경우 이 함수는 0 향해 자릅니다.

x 가 숫자가 아니거나 *base* 가 주어지면, *x* 는 문자열, *bytes*, 또는 *bytearray* 인스턴스여야 하는데, 진수 *base* 의 integer literal 을 나타내야 합니다. 선택적으로, 리터럴은 (사이에 공백 없이) `+` 또는 `-` 를 앞에 붙일 수 있고, 앞뒤로 공백에 둘러싸일 수 있습니다. 진수-n 리터럴은 0에서 n-1까지의 숫자로 구성되며, `a` 에서 `z` (또는 `A` 에서 `Z`) 가 10에서 35 사이의 값을 가집니다. 기본 *base* 는 10입니다. 허용되는 값은 0과 2–36입니다. 코드에서의 리터럴 처럼, 진수-2, -8 및 -16 리터럴에는 선택적으로 `0b`/`0B`, `0o`/`0O` 또는 `0x`/`0X` 접두사가 붙을 수 있습니다. base 0은 코드 리터럴과 똑같이 해석하라는 뜻이기 때문에, 실제 진수는 2, 8, 10 또는 16이고, 그래서 `int('010', 0)` 는 올바르지 않지만 `int('010', 8)` 뿐만 아니라 `int('010')` 도 올바릅니다.

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: *x*는 이제 위치 전용 매개 변수입니다.

버전 3.7.14에서 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 *x* to an *int* or when converting an *int* into a string would exceed the limit. See the *integer string conversion length limitation* documentation.

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), 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.

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, in which case every entry in *classinfo* will be checked. In any other case, a *TypeError* exception is raised.

iter(*object*[, *sentinel*])

이터레이터 객체를 돌려줍니다. 첫 번째 인자는 두 번째 인자의 존재 여부에 따라 매우 다르게 해석됩니다. 두 번째 인자가 없으면, *object* 는 이터레이션 프로토콜 (`__iter__()` 메서드)을 지원하는 컬렉션 객체이거나 시퀀스 프로토콜 (0에서 시작하는 정수 인자를 받는 `__getitem__()` 메서드)을 지원해야 합니다. 이러한 프로토콜 중 아무것도 지원하지 않으면 *TypeError* 가 일어납니다. 두 번째 인자 *sentinel* 이 주어지면, *object* 는 콜러블이어야 합니다. 이 경우 만들어지는 이터레이터는 *__next__()* 메서드가 호출될 때마다 인자 없이 *object* 를 호출합니다; 반환된 값이 *sentinel* 과 같으면, *StopIteration* 을 일으키고, 그렇지 않으면 값을 돌려줍니다.

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(*s*)
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.

class list([*iterable*])
함수이기보다, 리스트 와 시퀀스 형 — *list, tuple, range* 에 문서화 된 것처럼, `list` 는 실제로는 가변 시퀀스 형입니다.

locals()
현재 지역 심볼 테이블을 나타내는 딕셔너리를 갱신하고 돌려줍니다. `locals()` 이 함수 블록에서 호출될 때 자유 변수를 돌려주지만, 클래스 블록에서 호출할 때는 그렇지 않습니다. 모듈 수준에서 `locals()`와 `globals()`는 같은 딕셔너리임에 유의하십시오.

참고: 이 딕셔너리의 내용은 수정해서는 안 됩니다. 변경 사항은 인터프리터가 사용하는 지역 및 자유 변수의 값에 영향을 미치지 않을 수 있습니다.

---

map(*function*, *iterable*, ...)
iterable 의 모든 항목에 *function* 을 적용한 후 그 결과를 돌려주는 이터레이터를 돌려줍니다. 추가 *iterable* 인자가 전달되면, *function* 은 그 수 만큼의 인자를 받아들여야 하고 모든 이터러블에서 병렬로 제공되는 항목들에 적용됩니다. 다중 이터러블의 경우, 이터레이터는 가장 짧은 이터러블이 모두 소모되면 멈춥니다. 함수 입력이 이미 인자 튜플로 배치된 경우에는, `itertools.starmap()` 를 보세요.

max(*iterable*, *[, *key*, *default*])
max(*arg1*, *arg2*, **args*[, *key*])
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 3.4에 추가: *default* 키워드-전용 인자.

class memoryview(*obj*)
지정된 인자로부터 만들어진 "메모리 뷰" 객체를 돌려줍니다. 자세한 정보는 메모리 뷰 를 보세요.

min(*iterable*, *[, *key*, *default*])
min(*arg1*, *arg2*, **args*[, *key*])
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 3.4에 추가: *default* 키워드-전용 인자.

next (*iterator*[, *default*])
__next__() 메서드를 호출하여 *iterator* 에서 다음 항목을 꺼냅니다. *default* 가 주어지면, iterator가 고갈될 때 돌려주고, 그렇지 않으면 *StopIteration* 을 일으킵니다.

class object
새 기능 없는 객체를 돌려줍니다. *object* 는 모든 클래스의 베이스 클래스입니다. 모든 파이썬 클래스의 인스턴스에 공통적인 메서드를 가지고 있습니다. 이 함수는 인자를 받아들이지 않습니다.

참고: *object* 는 *__dict__* 을 가지지 않습니다. 그래서, *object* 클래스의 인스턴스에 임의의 어트리뷰트를 대입할 수 없습니다.

---

oct (*x*)
정수를 "0o"로 시작하는 8진수 문자열로 변환합니다. 결과는 올바른 파이썬 표현식입니다. *x* 가 파이썬 *int* 객체가 아니면, 정수를 돌려주는 __index__() 메서드를 정의해야 합니다. 예를 들어:

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

정수를 접두사 "0o"가 있거나 없는 형태의 8진수 문자열로 변환하려면, 다음 방법의 하나를 사용할 수 있습니다.

```
>>> '%#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.

file 은 열 파일의 경로명(절대 혹은 현재 작업 디렉터리에 상대적인)을 주는 경로류 객체 거나, 감쌀 파일의 정수 파일 디스크립터입니다. (파일 디스크립터가 주어지면, *closefd* 가 False 가 아닌 한, 반환된 I/O 객체가 닫힐 때 닫힙니다.)

mode 는 파일이 열리는 모드를 지정하는 선택적 문자열입니다. 기본값은 'r' 인데, 텍스트 모드로 읽기 위해 여는 것을 뜻합니다. 다른 일반적인 값은 쓰기 위한 'w' (파일이 이미 존재하는 경우 파일을 자릅니다), 독점적 파일 만들기를 위한 'x' 및 덧붙이기를 위한 'a' (일부 유닉스 시스템에서, 현재 위치와 관계없이 모든 쓰기가 파일의 끝에 덧붙여짐을 뜻합니다) 입니다. 텍스트 모드에서, *encoding* 을 지정하지 않으면 사용되는 인코딩은 플랫폼에 따라 다릅니다: 현재 로케일 인코딩을 얻기 위해 locale.getpreferredencoding(False) 가 호출됩니다. (날 바이트열을 읽고 쓰려면 바이너리 모드를 사용하고 *encoding* 을 지정하지 않습니다.) 사용 가능한 모드는 다음과 같습니다:

문자	의미
'r'	읽기용으로 엽니다 (기본값)
'w'	쓰기용으로 엽니다, 파일을 먼저 자릅니다.
'x'	독점적인 파일 만들기용으로 엽니다, 이미 존재하는 경우에는 실패합니다.
'a'	쓰기용으로 엽니다, 파일이 존재하는 경우는 파일의 끝에 덧붙입니다
'b'	바이너리 모드
't'	텍스트 모드 (기본값)
'+'	갱신 (읽기 및 쓰기) 용으로 디스크 파일을 엽니다

기본 모드는 'r' 입니다 (텍스트를 읽는 용으로 엽니다, 'rt' 의 동의어). 바이너리 읽기-쓰기 액세스의 경우는, 모드 'w+b' 는 파일을 열면서 0바이트로 자릅니다. 'r+b' 는 자르지 않고 파일을 엽니다.

Overview 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.

허용된 추가의 모드 문자 'U' 가 있습니다. 이것은 더는 아무런 효과가 없으며, 폐지된 것으로 간주합니다. 이전에는 텍스트 모드에서 유니버설 줄 넘김을 활성화했는데, 파이썬 3.0에서 기본 동작이 되었습니다. 자세한 내용은 *newline* 매개 변수의 설명서를 참조하십시오.

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

---

buffering 은 버퍼링 정책을 설정하는 데 사용되는 선택적 정수입니다. 버퍼링을 끄려면 (바이너리 모드에서만 허용) 0을 전달하고, 줄 버퍼링 (텍스트 모드에서만 사용 가능) 을 선택하려면 1을, 고정 크기 청크 버퍼를 선택하려면 그 크기를 바이트 단위로 표시한 정수 > 1을 전달합니다. *buffering* 인자가 제공되지 않을 때, 기본 버퍼링 정책은 다음과 같이 작동합니다:

- 바이너리 파일은 고정 크기 청크로 버퍼링 됩니다. 버퍼의 크기는 하부 장치의 "블록 크기"를 파악하려고 시도하는 경험적인 방법을 사용해서 선택되고 *io.DEFAULT_BUFFER_SIZE* 로 폴 백 됩니다. 많은 시스템에서, 버퍼는 일반적으로 4096 또는 8192바이트 길이입니다.
- "대화형" 텍스트 파일 (*isatty()* 가 True 를 돌려주는 파일) 은 줄 버퍼링을 사용합니다. 다른 텍스트 파일은 바이너리 파일에 대해 위에서 설명한 정책을 사용합니다.

encoding 은 파일을 디코딩하거나 인코딩하는 데 사용되는 인코딩의 이름입니다. 텍스트 모드에서만 사용해야 합니다. 기본 인코딩은 플랫폼에 따라 다르지만 (*locale.getpreferredencoding()* 이 돌려주는 값), 파이썬에서 지원하는 텍스트 인코딩 은 모두 사용할 수 있습니다. 지원되는 인코딩 목록은 *codecs* 모듈을 보면 됩니다.

errors 는 인코딩 및 디코딩 에러를 처리하는 방법을 지정하는 선택적 문자열입니다. 바이너리 모드에서는 사용할 수 없습니다. 다양한 표준 에러 처리기가 제공됩니다 (*Error Handlers* 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' 는 U+DC80에서 U+DCFF까지의 유니코드 개인 사용 영역의 코드 포인트로 잘못된 바이트를 나타냅니다. 데이터를 쓸 때 surrogateescape 에러 처리기가 사용되면, 이 개인 코드 포인트들은 원래의 바이트로 되돌아갑니다. 알 수 없는 인코딩의 파일을 처리할 때 유용합니다.

- 'xmlcharrefreplace' 는 파일에 쓸 때만 지원됩니다. 인코딩이 지원하지 않는 문자는 적절한 XML 문자 참조 &#nnn; 로 대체됩니다.
- '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 은 유니버설 줄 넘김 모드가 작동하는 방식을 제어합니다 (텍스트 모드에만 적용됩니다). 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. 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.

closefd 가 False 이고 파일명 대신 파일 디스크립터가 주어지면, 파일이 닫힐 때 하위 파일 디스크립터가 열려있게 됩니다. 파일명이 주어지면 *closefd* 는 True (기본값) 여야 합니다. 그렇지 않으면 에러가 발생합니다.

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.

fileinput, *io* (*open()* 이 선언된 곳), *os*, *os.path*, *tempfile*, 그리고 *shutil* 와 같은 파일 처리 모듈들도 보세요.

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.

Deprecated since version 3.4, will be removed in version 3.9: `'U'` 모드.

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.

ord(*c*)
하나의 유니코드 문자를 나타내는 문자열이 주어지면 해당 문자의 유니코드 코드 포인트를 나타내는 정수를 돌려줍니다. 예를 들어, `ord('a')` 는 정수 `97` 을 반환하고 `ord('€')` (유로 기호)는 `8364` 를 반환합니다. 이것은 *chr()* 의 반대입니다.

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

인자는 숫자 형이어야 합니다. 피연산자들의 형이 다를 경우, 이 항 산술 연산자에 대한 코어션 규칙이 적용됩니다. *int* 피연산자들의 경우, 결과는 두 번째 인자가 음수가 아닌 한 피연산자와 같은 형 (코어션 후에)이 됩니다; 두 번째 인자가 음수면 모든 인자가 float로 변환되고 float 결과가 전달됩니다. 예를 들어, `10**2` 는 `100` 을 반환하지만, `10**-2`` 는 ``0.01` 을 반환합니다. 두 번째 인자가 음수면 세 번째 인수는 생략해야 합니다. *z* 가 있는 경우, *x* 및 *y* 는 정수형이어야 하고, *y* 는 음수가 아니어야 합니다.

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)
objects 를 텍스트 스트림 *file* 로 인쇄하는데, *sep* 로 구분되고 *end* 를 뒤에 붙입니다. 있다면, *sep*, *end*, *file* 및 *flush* 는 반드시 키워드 인자로 제공해야 합니다.

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.

출력의 버퍼링 여부는 일반적으로 *file* 에 의해 결정되지만, *flush* 키워드 인자가 참이면 스트림이 강제로 플러시 됩니다.

버전 3.3에서 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

    def delx(self):
        del self._x

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

c 가 *C* 의 인스턴스면, `c.x` 는 게터(getter)를 호출하고, `c.x = value` 는 세터(setter)를 호출하고, `del c.x` 는 딜리터(deleter)를 호출합니다.

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
```

`@property` 데코레이터는 `voltage()` 메서드를 같은 이름의 읽기 전용 어트리뷰트에 대한 "게터"로 바꾸고, *voltage* 에 대한 독스트링을 "Get the current voltage." 로 설정합니다.

프로퍼티 객체는 데코레이터로 사용할 수 있는 `getter`, `setter` 및 `deleter` 메서드를 갖는데, 해당 접근자 함수를 데코레이트 된 함수로 설정한 프로퍼티의 사본을 만듭니다. 이것은 예제로 가장 잘 설명됩니다:

```
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.

class range(*stop*)
class range(*start*, *stop*[, *step*])
함수라기보다, *range* 는 실제로는 범위 와 시퀀스 형 — *list, tuple, range* 에 설명된 대로 불변 시퀀스 형입니다.

repr(*object*)
객체의 인쇄 가능한 표현을 포함한 문자열을 돌려줍니다. 많은 형에서, 이 함수는 *eval()* 에 전달 될 때 같은 값을 가진 객체를 생성하는 문자열을 반환하려고 시도합니다, 그렇지 않으면 표현은 객체의 형의 이름과 종종 객체의 이름과 주소를 포함하는 추가의 정보를 화살괄호로 묶은 문자열입니다. 클래스는 __repr__() 메서드를 정의하여 이 함수가 인스턴스에 대해 돌려주는 것을 제어할 수 있습니다.

reversed(*seq*)
역 이터레이터 를 돌려줍니다. *seq* 는 __reversed__() 메서드를 가졌거나 시퀀스 프로토콜(__len__() 메서드와 0 에서 시작하는 정수 인자를 받는 __getitem__() 메서드)을 지원하는 객체여야 합니다.

round(*number*[, *ndigits*])
number 를 소수점 다음에 *ndigits* 정밀도로 반올림한 값을 돌려줍니다. *ndigits* 가 생략되거나 None 이면, 입력에 가장 가까운 정수를 돌려줍니다.

round() 를 지원하는 내장형의 경우, 값은 10의 *-ndigits* 거듭제곱의 가장 가까운 배수로 반올림됩니다; 두 배수가 똑같이 가깝다면, 반올림은 짝수를 선택합니다(예를 들어, round(0.5) 와 round(-0.5) 는 모두 0 이고, round(1.5) 는 2 입니다). 모든 정숫값은 *ndigits* 에 유효합니다(양수, 0 또는 음수). *ndigits* 가 생략되거나 None 이면, 반환 값은 정수입니다. 그렇지 않으면 반환 값은 *number* 와 같은 형입니다.

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*)
이것은 *getattr()* 과 한 쌍입니다. 인자는 객체, 문자열 및 임의의 값입니다. 문자열은 기존 어트리뷰트 또는 새 어트리뷰트의 이름을 지정할 수 있습니다. 이 함수는 객체가 허용하는 경우 값을 어트리뷰트에 대입합니다. 예를 들어, setattr(x, 'foobar', 123) 는 x.foobar = 123 과 동등합니다.

class slice(*stop*)
class slice(*start*, *stop*[, *step*])
range(start, stop, step) 에 의해 지정된 인덱스 세트를 나타내는 슬라이스 객체를 돌려줍니다. *start* 및 *step* 인자의 기본값은 None 입니다. 슬라이스 객체는 단지 인자 값(또는 기본값)을 돌려주는 start, stop 및 step 의 읽기 전용 데이터 어트리뷰트를 갖습니다. 다른 명시적 기능은 없습니다; 그러나 Numerical Python과 다른 제삼자 확장이 사용합니다. 슬라이스 객체는 확장 인덱싱 문법을 사용할 때도 만들어집니다. 예를 들어: a[start:stop:step] 또는 a[start:stop, i]. 이터레이터를 돌려주는 대안 버전은 *itertools.islice()* 를 보세요.

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).

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, ...): ...
```

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

정적 메서드는 클래스 (`C.f()` 처럼) 또는 인스턴스 (`C().f()` 처럼)에 대해 호출할 수 있습니다.

파이썬의 정적 메서드는 자바 또는 C++ 에서 발견되는 정적 메서드와 비슷합니다. 대체 클래스 생성자를 만드는 데 유용한 변형을 보려면 *classmethod()* 도 보세요.

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:

```
class C:
    builtin_open = staticmethod(open)
```

정적 메서드에 대한 더 자세한 정보는, types을 참조하세요.

class str(*object=''*)
class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)
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*])
start 및 *iterable* 의 항목들을 왼쪽에서 오른쪽으로 합하고 합계를 돌려줍니다. *start* 의 기본값은 `0` 입니다. *iterable* 의 항목은 일반적으로 숫자며 시작 값은 문자열이 될 수 없습니다.

어떤 경우에는 *sum()* 에 대한 좋은 대안이 있습니다. 문자열의 시퀀스를 연결하는 가장 선호되고 빠른 방법은 `''.join(sequence)` 를 호출하는 것입니다. 확장된 정밀도로 부동 소수점 값을 더하려면 *math.fsum()* 를 보세요. 일련의 이터러블들을 연결하려면 *itertools.chain()* 를 고려해보세요.

super([*type*[, *object-or-type*]])
메서드 호출을 *type* 의 부모나 형제 클래스에 위임하는 프락시 객체를 돌려줍니다. 이는 클래스에서 재정의된 상속 된 메서드를 액세스할 때 유용합니다. 검색 순서는 *type* 자체를 건너뛰는 것을 제외하면, *getattr()* 에 의해 사용된 순서와 같습니다.

type 의 *__mro__* 어트리뷰트는 메서드 결정 검색 순서를 나열하는데 *getattr()* 과 *super()* 에서 사용됩니다. 이 어트리뷰트는 동적이며 상속 계층 구조가 변경될 때마다 바뀔 수 있습니다.

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).

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.

두 번째 사용 사례는 동적 실행 환경에서 협력적 다중 상속을 지원하는 것입니다. 이 사례는 파이썬에 고유하며 정적으로 컴파일되는 언어 또는 단일 상속만 지원하는 언어에서는 찾을 수 없습니다. 이것은 여러 베이스 클래스가 같은 메서드를 구현하는 "다이아몬드 다이어그램"을 구현할 수 있게 합니다. 좋은 설계는 모든 경우에 이 메서드가 같은 호출 시그니처를 갖도록 하는 것입니다(호출 순서는 실행 시간에 결정되기 때문에, 그 순서가 클래스 계층 구조의 변경에 적응하기 때문에, 그리고 그 순서가 실행 시간 전에 미리 알려지지 않은 형제 클래스를 포함할 수 있으므로).

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)
```

In addition to method lookups, *super()* also works for attribute lookups. One possible use case for this is calling *descriptor*s in a parent or sibling class.

super() 는 `super().__getitem__(name)` 과 같은 명시적인 점으로 구분된 어트리뷰트 조회를 위한 연결 절차의 일부로 구현됨에 주의하세요. 이것은 협력적인 다중 상속을 지원하는 예측 가능한 순서로 클래스를 검색하기 위해 자체 `__getattribute__()` 메서드를 구현함으로써 그렇게 합니다. 따라서, *super()* 는 `super()[name]` 과같이 문장이나 연산자를 사용하는 묵시적 조회에 대해서는 정의되지 않았습니다.

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.

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*)
인자 하나의 경우, *object* 의 형을 돌려줍니다. 반환 값은 형 객체며 일반적으로 *object.__class__* 가 돌려주는 것과 같은 객체입니다.

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

세 개의 인자를 주는 경우, 새 형 객체를 돌려줍니다. 이것은 본래 `class` 문의 동적인 형태입니다. *name* 문자열은 클래스 이름이고 *__name__* 어트리뷰트가 됩니다; *bases* 튜플은 베이스 클래스들을 항목화하고 *__bases__* 어트리뷰트가 됩니다; *dict* 딕셔너리는 클래스 바디의 정의들이 들어있는 이름 공간이며 *__dict__* 어트리뷰트가 되도록 표준 딕셔너리에 복사됩니다. 예를 들어, 다음 두 문장은 같은 *type* 객체를 만듭니다:

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

형 객체를 보세요.

버전 3.6에서 변경: type.__new__ 를 재정의하지 않는 *type* 의 서브 클래스는 이제 객체의 형을 얻기 위해 하나의 인자 형식을 사용할 수 없습니다.

vars([*object*])

모듈, 클래스, 인스턴스 또는 *__dict__* 어트리뷰트가 있는 다른 객체의 *__dict__* 어트리뷰트를 돌려줍니다.

모듈 및 인스턴스와 같은 객체는 업데이트 가능한 *__dict__* 어트리뷰트를 갖습니다; 그러나, 다른 객체는 *__dict__* 어트리뷰트에 쓰기 제한을 가질 수 있습니다 (예를 들어, 클래스는 직접적인 딕셔너리 갱신을 방지하기 위해 *types.MappingProxyType* 를 사용합니다).

인자가 없으면, *vars()* 는 *locals()* 처럼 동작합니다. locals 딕셔너리에 대한 변경이 무시되기 때문에 locals 딕셔너리는 읽기에만 유용하다는 것에 주의하세요.

zip(**iterables*)

각 iterables 의 요소들을 모으는 이터레이터를 만듭니다.

튜플의 이터레이터를 돌려주는데, *i* 번째 튜플은 각 인자로 전달된 시퀀스나 이터러블의 *i* 번째 요소를 포함합니다. 이터레이터는 가장 짧은 입력 이터러블이 모두 소모되면 멈춥니다. 하나의 이터러블 인자를 사용하면, 1-튜플의 이터레이터를 돌려줍니다. 인자가 없으면, 빈 이터레이터를 돌려줍니다. 다음과 동등합니다:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

iterables 를 왼쪽에서 오른쪽으로 값을 구하는 순서가 보장됩니다. 이것은 zip(*[iter(s)]*n) 을 사용하여 데이터 시리즈를 길이 n인 그룹으로 클러스터링하는 관용구를 가능하게 만듭니다. 이것은 같은 이터레이터를 n 번 반복해서, 각 출력 튜플이 이터레이터를 n 번 호출한 결과를 갖게 됩니다. 입력을 길이 n인 묶음으로 나누는 효과를 줍니다.

zip() 에 길이가 같지 않은 입력들을 제공하는 것은, 끝부분에서 매치되지 않고 남는 더 긴 이터러블들의 값들에 신경 쓰지 않는 경우로 제한해야 합니다. 그 값들이 중요하다면, 대신 *itertools.zip_longest()* 를 사용하세요.

zip() 을 * 연산자와 함께 쓰면 리스트를 unzip 할 수 있습니다:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> x == list(x2) and y == list(y2)
True
```

__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.

함수는 모듈 *name* 을 임포트 하는데, 잠재적으로 패키지 문맥에서 이름을 해석하는 방법을 결정하는데 주어진 *globals* 와 *locals* 를 사용합니다. *fromlist* 는 *name* 에 의해 주어진 모듈로부터 임포트 되어야 하는 객체 또는 서브 모듈의 이름을 제공합니다. 표준 구현은 *locals* 인자를 전혀 사용하지 않고, import 문의 패키지 문맥을 결정할 때만 *globals* 를 사용합니다.

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).

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
NoneType 형의 유일한 값. None 은 기본 인자가 함수에 전달되지 않을 때처럼, 값의 부재를 나타내는 데 자주 사용됩니다. None 에 대입할 수 없고 *SyntaxError* 를 일으킵니다.

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. 논릿값은 참입니다.

> **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.

> **참고:** NotImplementedError 와 NotImplemented 는 비슷한 이름과 목적이 있지만, 바꿔쓸 수 없습니다. 언제 사용하는지 자세히 알고 싶다면 *NotImplementedError*를 보세요.

Ellipsis
Ellipsis 리터럴 "..." 와 같습니다. 주로 사용자 정의 컨테이너 데이터형에 대한 확장 슬라이스 문법과 함께 사용되는 특수 값.

__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*)
인쇄될 때, "Use quit() or Ctrl-D (i.e. EOF) to exit"과 같은 메시지를 인쇄하고, 호출될 때, 지정된 종료 코드로 *SystemExit* 를 일으키는 객체.

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 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.

기본적으로 객체는 클래스가 그 객체에 대해 호출될 때 `False` 를 돌려주는 `__bool__()` 메서드나 `0`을 돌려주는 `__len__()` 메서드를 정의하지 않는 한 참으로 간주합니다.[1] 여기에 거짓으로 간주하는 대부분의 내장 객체들이 있습니다:

- 거짓으로 정의된 상수: `None` 과 `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.)

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

4.2 논리 연산 — and, or, not

이것들은 우선순위에 따라 오름차순으로 정렬된 논리 연산들입니다:

연산	결과	노트
`x or y`	*x* 가 거짓이면 *y*, 그렇지 않으면 *x*	(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 8가지 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. 더 나아가, 어떤 형들은 (예를 들어, 함수 객체) 그 형의 모든 두 객체가 다르다고 비교되는 비교의 축약적인 개념만을 지원합니다. `<`, `<=`, `>`, `>=` 연산자들은 복소수를 다른 내장 숫자 형과 비교할 때, 객체들이 비교될 수 없는 다른 형일 때, 정의된 순서가 없을 때 *TypeError* 예외를 일으킵니다.

클래스의 같지 않은 인스턴스들은 그 클래스가 `__eq__()` 메서드를 정의하지 않는 이상 보통 같지 않다고 비교됩니다.

클래스가 `__lt__()`, `__le__()`, `__gt__()`, `__ge__()` 메서드들을 충분히 정의하지 않는 이상, 클래스의 인스턴스들은 같은 클래스의 다른 인스턴스나 다른 형의 객체와의 순서가 정해지지 않습니다 (일반적으로, 여러분이 비교 연산자의 관습적인 의미를 원한다면 `__lt__()` 와 `__eq__()` 만으로 충분합니다).

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

같은 문법적 우선순위를 갖는 두 개의 연산, `in` 과 `not in`, 은 이터러블이거나 `__contains__()` 메서드를 구현하는 형에서 지원됩니다.

4.4 숫자 형 — int, float, complex

세 가지 다른 숫자 형이 있습니다: 정수 *(integers)*, 실수 *(floating point numbers)*, 복소수 *(complex numbers)*. 또한 논리형은 정수의 하위 유형입니다. 정수는 무제한의 정밀도를 갖습니다. 실수는 보통 C의 `double`을 사용해서 구현됩니다; 프로그램이 실행되고 있는 기계의 부동 소수점 숫자의 정밀도와 내부 표현에 관한 정보는 *sys.float_info*에서 얻을 수 있습니다. 복소수는 각각 실수로 표현되는 실수부와 허수부를 가집니다. 복소수 *z* 에서 이들 부분을 추출하려면 `z.real` 과 `z.imag` 를 사용하십시오. (표준 라이브러리는 추가적인 숫자 형들을 포함하는데, *fractions*는 유리수를, *decimal* 은 사용자가 정의할 수 있는 정밀도로 부동 소수점 숫자를 다룹니다.)

숫자는 숫자 리터럴 또는 내장 함수와 연산자의 결과로 만들어집니다. 꾸밈없는 정수 리터럴(16진수, 8진수, 2진수 포함)은 정수를 만듭니다. 소수점 또는 지수 기호가 포함된 숫자 리터럴은 실수를 만듭니다. 숫자 리터럴에 `'j'` 나 `'J'` 를 덧붙이면 허수 (실수부가 0인 복소수) 가 만들어지는데, 정수나 실수에 더해서 실수부와 허수부가 있는 복소수를 만들 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, which is narrower than complex. A comparison between numbers of different types behaves as though the exact values of those numbers were being compared.[2]

The constructors *int()*, *float()*, and *complex()* can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations (for priorities of the operations, see operator-summary):

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)	
`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 0입니다.	(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) 정수 나눗셈이라고 도합니다. 결괏값의 형이 꼭 int 일 필요는 없지만, 결괏값은 항상 정수입니다. 결과는 항상 음의 무한대를 향해 내림 됩니다: `1//2` 는 `0`, `(-1)//2` 는 `-1`, `1//(-2)` 는 `-1`, `(-1)//(-2)` 는 `0` 입니다.

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

(3) 실수에서 정수로의 변환은 C에서처럼 반올림이나 자름이 발생할 수 있습니다; 잘 정의된 변환을 위해서는 *math.floor()* 와 *math.ceil()* 함수를 보십시오.

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

(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.

(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.

`Nd` 속성을 가진 코드 포인트의 전체 목록을 보려면 http://www.unicode.org/Public/10.0.0/ucd/extracted/DerivedNumericType.txt 를 보십시오.

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 0 입니다.
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) A left shift by *n* bits is equivalent to multiplication by `pow(2, n)`.

(3) A right shift by *n* bits is equivalent to floor division by `pow(2, n)`.

(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 클래스를 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
```

버전 3.1에 추가.

int.**to_bytes**(*length*, *byteorder*, *, *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'
```

정수는 *length* 바이트를 사용하여 표현됩니다. 정수가 주어진 바이트 수로 표현할 수 없는 경우 An *OverflowError* 를 일으킵니다.

byteorder 인자는 정수를 나타내는 데 사용되는 바이트 순서를 결정합니다. *byteorder* 가 `"big"` 인 경우, 최상위 바이트는 바이트 배열의 처음에 있습니다. *byteorder* 가 `"little"` 인 경우, 최상위 바이트는 바이트 배열의 끝에 있습니다. 호스트 시스템의 기본 바이트 순서를 요청하려면 바이트 순서 값으로 *sys.byteorder* 를 사용하십시오.

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

버전 3.2에 추가.

classmethod int.**from_bytes**(*bytes*, *byteorder*, *, *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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.

byteorder 인자는 정수를 나타내는 데 사용되는 바이트 순서를 결정합니다. *byteorder* 가 `"big"` 인 경우, 최상위 바이트는 바이트 배열의 처음에 있습니다. *byteorder* 가 `"little"` 인 경우, 최상위 바이트는 바이트 배열의 끝에 있습니다. 호스트 시스템의 기본 바이트 순서를 요청하려면 바이트 순서 값으로 *sys.byteorder* 를 사용하십시오.

signed 인자는 정수를 표현하는데 2의 보수가 사용되는지를 나타냅니다.

버전 3.2에 추가.

4.4.3 실수에 대한 추가 메서드

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

`float.`**`as_integer_ratio`**`()`
: 비율이 원래 float와 정확히 같고 양의 분모를 갖는 정수 쌍을 돌려줍니다. 무한대에는 *OverflowError* 를, NaN 에는 a *ValueError* 를 일으킵니다.

`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 숫자 형의 해싱

숫자 `x` 와 `y`, 서로 다른 형이어도 됩니다, 에 대하여, `x == y` 면 항상 `hash(x) == hash(y)` 일 것이 요구됩니다 (자세한 내용은 `__hash__()` 메서드 설명서를 보세요). 다양한 숫자 형 (*int*, *float*, *decimal.Decimal*, *fractions.Fraction* 포함) 들의 구현의 편의성과 효율 때문에, 파이썬의 숫자 형의 해시는 단일한 수학 함수에 기반을 두고 있고, 이 함수는 임의의 유리수에 대해 정의되어서 *int* 와 *fractions.Fraction* 의 모든 인스턴스, *float* 와 *decimal.Decimal* 의 모든 유한 인스턴스에 적용됩니다. 본질에서, 이 함수는 고정 소수 `P` 에 대해 모듈로 `P` 환원 (reduction modulo `P`) 으로 주어집니다. `P` 의 값은 *sys.hash_info* 의 `modulus` 어트리뷰트로 파이썬에 제공됩니다.

CPython implementation detail: 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 나뉘지 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` 나뉘면 (하지만 `m` 은 나뉘지 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.
- 특별한 값 `sys.hash_info.inf`, `-sys.hash_info.inf`, `sys.hash_info.nan` 은 각각 무한대, 음의 무한대, nan 으로 사용됩니다. (모든 해시 가능 nan 은 같은 해시값을 가집니다.)
- 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)).
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    """
    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 sys.hash_info.nan
    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 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.

컨테이너 객체가 이터레이션 지원을 제공하려면 한가지 메서드를 정의할 필요가 있습니다.:

container.**__iter__**()
: 이터레이터 객체를 돌려줍니다. 이 객체는 아래에서 설명하는 이터레이터 프로토콜을 지원해야 합니다. 컨테이너가 여러 유형의 이터레이션을 지원하는 경우, 이터레이션 유형에 대한 이터레이터를 구체적으로 요구하는 추가 메서드를 제공할 수 있습니다. (여러 형태의 이터레이션을 지원하는 객체의 예로 너비 우선과 깊이 우선 탐색을 모두 지원하는 트리 구조를 들 수 있습니다.) 이 메서드는 파이썬/C API에서 파이썬 객체를 위한 구조체의 `tp_iter` 슬롯에 대응합니다.

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

iterator.**__iter__**()
: 이터레이터 객체 자신을 돌려줍니다. 이는 `for` 와 `in` 문에 컨테이너와 이터레이터 모두 사용될 수 있게 하는 데 필요합니다. 이 메서드는 파이썬/C API에서 파이썬 객체를 위한 구조체의 `tp_iter` 슬롯에 대응합니다.

iterator.**__next__**()
: 컨테이너의 다음 항목을 돌려줍니다. 더 항목이 없으면 *StopIteration* 예외를 일으킵니다. 이 메서드는 파이썬/C API에서 파이썬 객체를 위한 구조체의 `tp_iternext` 슬롯에 대응합니다.

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.

4.5.1 제너레이터 형

파이썬의 제너레이터 는 이터레이터 프로토콜을 구현하는 편리한 방법을 제공합니다. 컨테이너 객체의 `__iter__()` 메서드가 제너레이터로 구현되면, `__iter__()` 와 `__next__()` 메서드를 제공하는 이터레이터 객체(기술적으로, 제너레이터 객체)를 자동으로 돌려줍니다. 제너레이터에 대한 더 자세한 정보는 일드 표현식 설명서 에서 찾을 수 있습니다.

4.6 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.

4.6.1 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]

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

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)
`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	
`s.index(x[, i[, j]])`	(인덱스 *i* 또는 그 이후에, 인덱스 *j* 전에 등장하는) *s* 의 첫 번째 *x* 의 인덱스	(8)
`s.count(x)`	*s* 등장하는 *x* 의 총수	

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.)

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:

```
>>> 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) *s* 에 *x* 가 없을 때 index 는 *ValueError* 를 일으킵니다. 모든 구현이 추가 인자 *i* 및 *j* 전달을 지원하지는 않습니다. 이러한 인자를 사용하면 시퀀스의 부분을 효율적으로 검색할 수 있습니다. 추가 인자를 전달하는 것은 대략 `s[i:j].index(x)` 를 사용하는 것과 비슷한데, 데이터를 복사하지 않고 반환된 인덱스가 슬라이스의 시작이 아닌 시퀀스의 시작을 기준으로 삼습니다.

4.6.2 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.

4.6.3 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	
`s[i:j] = t`	*i* 에서 *j* 까지의 *s* 슬라이스가 이터러블 *t* 의 내용으로 대체됩니다	
`del s[i:j]`	`s[i:j] = []` 와 같습니다	
`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.append(x)`	시퀀스의 끝에 *x* 를 추가합니다 (`s[len(s):len(s)] = [x]` 와 같습니다)	
`s.clear()`	*s* 에서 모든 항목을 제거합니다 (`del s[:]` 와 같습니다)	(5)
`s.copy()`	*s* 의 얕은 복사본을 만듭니다 (`s[:]` 와 같습니다)	(5)
`s.extend(t)` 또는 `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	(6)
`s.insert(i, x)`	*x* 를 *s* 의 *i* 로 주어진 인덱스에 삽입합니다 (`s[i:i] = [x]` 와 같습니다)	
`s.pop([i])`	*i* 에 있는 항목을 꺼냄과 동시에 *s* 에서 제거합니다	(2)
`s.remove(x)`	`s[i]` 가 *x* 와 같은 첫 번째 항목을 *s* 에서 제거합니다	(3)
`s.reverse()`	제자리에서 *s* 의 항목들의 순서를 뒤집습니다	(4)

노트:

(1) *t* 는 교체할 슬라이스와 길이가 같아야 합니다.

(2) 선택적 인자 *i* 의 기본값은 `-1` 입니다. 그래서 기본적으로 마지막 항목이 제거되면서 반환됩니다.

(3) *x* 가 *s* 에서 발견되지 않으면 `remove` 는 *ValueError* 를 일으킵니다.

(4) 큰 시퀀스를 뒤집을 때 공간 절약을 위해 `reverse()` 메서드는 제자리에서 시퀀스를 수정합니다. 부작용으로 작동한다는 것을 사용자에게 상기시키기 위해 뒤집힌 시퀀스를 돌려주지 않습니다.

(5) `clear()` 와 `copy()` 는 슬라이싱 연산을 지원하지 않는 (*dict* 와 *set* 같은) 가변 컨테이너들의 인터페이스와 일관성을 유지하기 위해 포함됩니다

버전 3.3에 추가: `clear()` 와 `copy()` 메서드.

(6) *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.

4.6.4 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).

CPython implementation detail: 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.

4.6.5 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.

4.6.6 범위

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

class range(*stop*)
class range(*start*, *stop*[, *step*])

범위 생성자에 대해 인자는 정수여야 합니다(내장 *int* 또는 `__index__` 특수 메서드를 구현하는 임의의 객체). *step* 인자가 생략되면 기본값 `1` 이 사용됩니다. *start* 인자가 생략되면 기본값 `0` 이 사용됩니다. *step* 이 0이면 *ValueError* 를 일으킵니다.

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.

버전 3.3에 추가: *start*, *stop*, *step* 어트리뷰트.

더 보기:

- linspace recipe 에서는 부동 소수점 응용 프로그램에 적합한 범위의 지연된 버전을 구현하는 방법을 보여줍니다.

4.7 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'`
- 큰따옴표: `"'작은' 따옴표를 담을 수 있습니다"`.
- 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"`.

지원되는 이스케이프 시퀀스와 대부분의 이스케이프 시퀀스 처리를 비활성화하는 `r` ("날") 접두어를 포함하여 문자열 리터럴의 다양한 형식에 대한 자세한 내용은 strings 을 참조하십시오.

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

별도의 "문자" 형이 없으므로 문자열을 인덱싱하면 길이가 1인 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 3.3에서 변경: 파이썬 2시리즈와의 하위 호환성을 위해서, `u` 접두어가 문자열 리터럴에 다시 한번 허용됩니다. 문자열 리터럴의 의미에 영향을 미치지 않으며 `r` 접두사와 결합 될 수 없습니다.

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

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.

encoding 과 *errors* 모두 주어지지 않으면, `str(object)` 는 `object.__str__()` 를 돌려주는데, *object* 의 "비형식적" 또는 멋지게 인쇄 가능한 문자열 표현입니다. 문자열 객체의 경우, 이것은 문자열 자신입니다. 만약 *object* 가 `__str__()` 메서드를 가지고 있지 않다면, *str()* 은 대신 *repr(object)* 를 돌려줍니다.

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.

4.7.1 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.

`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.

케이스 폴딩 알고리즘은 유니코드 표준의 섹션 3.13 에 설명되어 있습니다.

버전 3.3에 추가.

str.**center**(*width*[, *fillchar*])
길이 *width* 인 문자열의 가운데에 정렬한 값을 돌려줍니다. 지정된 *fillchar* (기본값은 ASCII 스페이스)을 사용하여 채웁니다. *width* 가 `len(s)` 보다 작거나 같은 경우 원래 문자열이 반환됩니다.

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.

str.**encode**(*encoding="utf-8"*, *errors="strict"*)
문자열의 바이트열 객체로 인코딩된 버전을 돌려줍니다. 기본 인코딩은 `'utf-8'` 입니다. *errors* 는 다른 오류 처리 방식을 설정하기 위해 제공될 수 있습니다. *errors* 의 기본값은 `'strict'` 인데, 인코딩 오류가 있으면 *UnicodeError* 를 일으키라는 뜻입니다. 다른 가능한 값은 `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` 와 *codecs.register_error()* 를 통해 등록된 다른 이름들입니다. *Error Handlers*를 보세요. 가능한 인코딩의 목록을 보려면 *Standard Encodings* 섹션을 참조하십시오.

버전 3.1에서 변경: 키워드 인자 지원이 추가되었습니다.

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

str.**expandtabs**(*tabsize=8*)
모든 탭 문자들을 현재의 열과 주어진 탭 크기에 따라 하나나 그 이상의 스페이스로 치환한 문자열의 복사본을 돌려줍니다. 탭 위치는 *tabsize* 문자마다 발생합니다 (기본값은 8이고, 열 0, 8, 16 등에 탭 위치를 지정합니다). 문자열을 확장하기 위해 현재 열이 0으로 설정되고 문자열을 문자 단위로 검사합니다. 문자가 탭 (`\t`) 이면, 현재 열이 다음 탭 위치와 같아질 때까지 하나 이상의 스페이스 문자가 삽입됩니다. (탭 문자 자체는 복사되지 않습니다.) 문자가 개행 문자 (`\n`) 또는 캐리지 리턴 (`\r`) 이면 복사되고 현재 열은 0으로 재설정됩니다. 다른 문자는 변경되지 않고 복사되고 현재 열은 인쇄할 때 문자가 어떻게 표시되는지에 관계없이 1씩 증가합니다.

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

str.**find**(*sub*[, *start*[, *end*]])
부분 문자열 *sub* 가 슬라이스 `s[start:end]` 내에 등장하는 가장 작은 문자열의 인덱스를 돌려줍니다. 선택적 인자 *start* 와 *end* 는 슬라이스 표기법으로 해석됩니다. *sub* 가 없으면 `-1` 을 돌려줍니다.

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'
```

버전 3.2에 추가.

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

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

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

Return `True` if all characters in the string are alphanumeric and there is at least one character, `False` otherwise. A character `c` is alphanumeric if one of the following returns `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `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 Unicode Standard.

`str.`**`isascii`**`()`

Return `True` if the string is empty or all characters in the string are ASCII, `False` otherwise. ASCII characters have code points in the range U+0000-U+007F.

버전 3.7에 추가.

`str.`**`isdecimal`**`()`

Return `True` if all characters in the string are decimal characters and there is at least one character, `False` otherwise. Decimal characters are those that can be used to form numbers in base 10, e.g. U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category “Nd”.

`str.`**`isdigit`**`()`

Return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

str.**isidentifier**()
Return `True` if the string is a valid identifier according to the language definition, section identifiers.

`def` 나 `class`와 같은 예약 식별자를 검사하려면 *keyword.iskeyword()* 를 사용하십시오.

str.**islower**()
Return `True` if all cased characters[4] in the string are lowercase and there is at least one cased character, `False` otherwise.

str.**isnumeric**()
Return `True` if all characters in the string are numeric characters, and there is at least one character, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

str.**isprintable**()
Return `True` if all characters in the string are printable or the string is empty, `False` otherwise. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when *repr()* is invoked on a string. It has no bearing on the handling of strings written to *sys.stdout* or *sys.stderr*.)

str.**isspace**()
Return `True` if there are only whitespace characters in the string and there is at least one character, `False` otherwise.

A character is *whitespace* if in the Unicode character database (see *unicodedata*), either its general category is `Zs` ("Separator, space"), or its bidirectional class is one of `WS`, `B`, or `S`.

str.**istitle**()
Return `True` if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return `False` otherwise.

str.**isupper**()
Return `True` if all cased characters[4] in the string are uppercase and there is at least one cased character, `False` otherwise.

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[4] 가 소문자로 변환된 문자열의 복사본을 돌려줍니다.

사용되는 소문자 변환 알고리즘은 유니코드 표준의 섹션 3.13에 설명되어 있습니다.

str.**lstrip**([*chars*])
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'
```

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

static str.**maketrans**(*x*[, *y*[, *z*]])
이 정적 메서드는 *str.translate()* 에 사용할 수 있는 변환표를 돌려줍니다.

인자가 하나만 있으면 유니코드 포인트(정수) 또는 문자(길이가 1인 문자열)를 유니코드 포인트, 문자열(임의 길이) 또는 None 으로 매핑하는 딕셔너리여야 합니다. 문자 키는 유니코드 포인트로 변환됩니다.

인자가 두 개면 길이가 같은 문자열이어야 하며, 결과 딕셔너리에서, x의 각 문자는 y의 같은 위치에 있는 문자로 대응됩니다. 세 번째의 인자가 있는 경우, 문자열이어야 하는데 각 문자가 None 으로 대응되는 결과를 줍니다.

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.**replace**(*old*, *new*[, *count*])
모든 부분 문자열 *old* 가 *new* 로 치환된 문자열의 복사본을 돌려줍니다. 선택적 인자 *count* 가 주어지면, 앞의 *count* 개만 치환됩니다.

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*])
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'
```

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).

sep 이 주어지면, 연속된 구분자는 묶이지 않고 빈 문자열을 구분하는 것으로 간주합니다(예를 들어, '1,,2'.split(',') 는 ['1', '', '2'] 를 돌려줍니다). *sep* 인자는 여러 문자로 구성될 수 있습니다(예를 들어, '1<>2<>3'.split('<>') 는 ['1', '2', '3'] 를 돌려줍니다). 지정된 구분자로 빈 문자열을 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', '']
```

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']
```

str.**splitlines**([*keepends*])

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()
[]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> "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*])
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"
```

정규식을 사용하여 아포스트로피에 대한 해결 방법을 구성할 수 있습니다:

```
>>> import re
>>> def titlecase(s):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

str.**translate**(*table*)

각 문자를 지정된 변환표를 사용해 매핑한 문자열의 복사본을 돌려줍니다. table은 `__getitem__()` 을 통한 인덱싱을 구현하는 객체여야 하는데, 보통 매핑 이나 시퀀스 입니다. 유니코드 포인트 (정수) 로 인덱싱할 때, table 객체는 다음 중 하나를 수행할 수 있습니다: 그 문자를 하나 이상의 다른 문자들로 매핑하기 위해 유니코드 포인트나 문자열을 돌려줍니다; 결과 문자열에서 그 문자를 제거하기 위해 None 을 돌려줍니다; 그 문자를 자기 자신으로 매핑하기 위해 *LookupError* 예외를 일으킵니다.

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[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.

사용되는 대문자 변환 알고리즘은 유니코드 표준의 섹션 3.13에 설명되어 있습니다.

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'
```

4.7.2 printf 스타일 문자열 포매팅

참고: 여기에 설명된 포맷 연산은 여러 가지 일반적인 오류를 (예를 들어 튜플과 딕셔너리를 올바르게 표시하지 못하는 것) 유발하는 다양한 문제점들이 있습니다. 새 포맷 문자열 리터럴 나 *str.format()* 인터페이스 혹은 템플릿 문자열 을 사용하면 이러한 오류를 피할 수 있습니다. 이 대안들은 또한 텍스트 포매팅에 더욱 강력하고 유연하며 확장 가능한 접근법을 제공합니다.

문자열 객체는 한가지 고유한 내장 연산을 갖고 있습니다: `%` 연산자 (모듈로). 이것은 문자열 포매팅 또는 치환 연산자라고도 합니다. `format % values` 가 주어질 때 (*format* 은 문자열입니다), *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.[5] 그렇지 않으면, *values* 는 format 문자열이 지정하는 항목의 수와 같은 튜플이거나 단일 매핑 객체 (예를 들어, 딕셔너리) 이어야 합니다.

[5] 그래서, 튜플만을 포매팅하려면 포맷할 튜플 하나만을 포함하는 1-튜플을 제공해야 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 매핑으로 부터 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:

플 래 그	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.	(6)
'x'	부호 있는 16진수 (소문자).	(2)
'X'	부호 있는 16진수 (대문자).	(2)
'e'	부동 소수점 지수 형식 (소문자).	(3)
'E'	부동 소수점 지수 형식 (대문자).	(3)
'f'	부동 소수점 십진수 형식.	(3)
'F'	부동 소수점 십진수 형식.	(3)
'g'	부동 소수점 형식. 지수가 -4보다 작거나 정밀도 보다 작지 않으면 소문자 지수형식을 사용하고, 그렇지 않으면 십진수 형식을 사용합니다.	(4)
'G'	부동 소수점 형식. 지수가 -4보다 작거나 정밀도 보다 작지 않으면 대문자 지수형식을 사용하고, 그렇지 않으면 십진수 형식을 사용합니다.	(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.

4.8 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.

4.8.1 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*[, *encoding*[, *errors*]]])

첫째로, 바이트열 리터럴의 문법은 문자열 리터럴과 거의 같지만 b 접두사가 추가된다는 점이 다릅니다.:

- 작은따옴표: `b'still allows embedded "double" quotes'`
- 큰따옴표: `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.

2개의 16진수는 정확히 하나의 바이트에 대응하기 때문에 16진수는 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 16진수가 포함되어야 하며 ASCII 공백은 무시됩니다.

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

버전 3.7에서 변경: 이제 *bytes.fromhex()* 는 스페이스뿐만 아니라 문자열에 있는 모든 ASCII 공백을 건너뜁니다.

바이트열 객체를 16진수 표현으로 변환하기 위한 역변환 함수가 있습니다.

hex()

인스턴스의 바이트마다 2 자릿수의 16진수로 표현한 문자열 객체를 돌려줍니다.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

버전 3.5에 추가.

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.

참고: 파이썬 2.x 사용자에게: 파이썬 2.x 시리즈에서는 8-비트 문자열(2.x가 내장 바이너리 데이터형에 제공하는 가장 가까운 것)과 유니코드 문자열 간의 다양한 묵시적 변환이 허용되었습니다. 이는 파이썬이 원래 8-비트 텍스트만 지원했으며 유니코드 텍스트는 나중에 추가된 사실을 반영하는 하위 호환성 해결책입니다. 파이썬 3.x 에서, 이러한 묵시적 변환은 사라졌습니다 - 8-비트 바이너리 데이터와 유니코드 텍스트 간의 변환은 반드시 명시적이어야 하며 바이트열과 문자열 객체는 항상 다르다고 비교됩니다.

4.8.2 바이트 배열 객체

bytearray 객체는 *bytes* 객체의 가변형입니다.

class bytearray([*source*[, *encoding*[, *errors*]]])

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.

바이트 배열 객체를 16진수 표현으로 변환하기 위한 역변환 함수가 있습니다.

hex()

인스턴스의 바이트마다 2 자릿수의 16진수로 표현한 문자열 객체를 돌려줍니다.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

버전 3.5에 추가.

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.

4.8.3 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.

버전 3.3에서 변경: 서브 시퀀스로 0에서 255 사이의 정수도 허용합니다.

`bytes.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)
`bytearray.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)

주어진 바이트열로부터 디코딩된 문자열을 돌려줍니다. 기본 인코딩은 `'utf-8'` 입니다. *errors* 는 다른 오류 처리 방식을 설정하기 위해 제공될 수 있습니다. *errors* 의 기본값은 `'strict'` 인데, 인코딩 오류가 있으면 `UnicodeError` 를 일으키라는 뜻입니다. 다른 가능한 값은 `'ignore'`, `'replace'` 와 `codecs.register_error()` 를 통해 등록된 다른 이름들입니다. *Error Handlers*를 보세요. 가능한 인코딩의 목록을 보려면 *Standard Encodings* 섹션을 참조하십시오.

참고: *encoding* 인자를 `str` 에 전달하면 임시 바이트열이나 바이트 배열 객체를 만들 필요 없이 임의의 *bytes-like object* 를 직접 디코딩할 수 있습니다.

버전 3.1에서 변경: 키워드 인자 지원이 추가되었습니다.

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.

버전 3.1에 추가.

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*])
bytearray.**replace**(*old*, *new*[, *count*])
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*])
bytearray.**center**(*width*[, *fillbyte*])
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*])
`bytearray.`**`ljust`**(*width*[, *fillbyte*])
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`**([*chars*])
`bytearray.`**`lstrip`**([*chars*])
선행 바이트가 제거된 시퀀스의 복사본을 돌려줍니다. *chars* 인자는 제거할 바이트 집합을 지정하는 바이너리 시퀀스입니다 - 이름은 이 메서드가 보통 ASCII 문자와 사용된다는 사실을 반영합니다. 생략되거나 `None` 이라면, *chars* 인자의 기본값은 ASCII 공백을 제거하도록 합니다. *chars* 인자는 접두사가 아닙니다; 모든 값 조합이 제거됩니다:

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

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.

`bytes.`**`rjust`**(*width*[, *fillbyte*])
`bytearray.`**`rjust`**(*width*[, *fillbyte*])
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`**([*chars*])
`bytearray.`**`rstrip`**([*chars*])
지정된 후행 바이트가 제거된 시퀀스의 복사본을 돌려줍니다. *chars* 인자는 제거할 바이트 집합을 지정

하는 바이너리 시퀀스입니다 - 이름은 이 메서드가 보통 ASCII 문자와 사용된다는 사실을 반영합니다. 생략되거나 None 이라면, *chars* 인자의 기본값은 ASCII 공백을 제거하도록 합니다. *chars* 인자는 접미사가 아닙니다; 모든 값 조합이 제거됩니다:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

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.

---

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).

sep 이 주어지면, 연속된 구분자는 묶이지 않고 빈 서브 시퀀스를 구분하는 것으로 간주합니다 (예를 들어, b'1,,2'.split(b',') 는 [b'1', b'', b'2'] 를 돌려줍니다). *sep* 인자는 멀티바이트 시퀀스로 구성될 수 있습니다 (예를 들어, b'1<>2<>3'.split(b'<>') 는 [b'1', b'2', b'3'] 를 돌려줍니다). 지정된 구분자로 빈 시퀀스를 나누면, 나누는 객체의 형에 따라 [b''] 나 [bytearray(b'')] 를 돌려줍니다. *sep* 인자는 임의의 *bytes-like object* 일 수 있습니다.

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'']
```

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**([*chars*])
bytearray.**strip**([*chars*])

선행과 후행 바이트가 제거된 시퀀스의 복사본을 돌려줍니다. *chars* 인자는 제거할 바이트 집합을 지정하는 바이너리 시퀀스입니다 - 이름은 이 메서드가 보통 ASCII 문자와 사용된다는 사실을 반영합니다. 생략되거나 None 이라면, *chars* 인자의 기본값은 ASCII 공백을 제거하도록 합니다. *chars* 인자는 접두사나 접미사가 아닙니다; 모든 값 조합이 제거됩니다:

```
>>> b'   spacious   '.strip()
b'spacious'
>>> 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**()
Return `True` if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

예를 들면:

```
>>> b'ABCabc1'.isalnum()
True
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()
bytearray.**isalpha**()

Return `True` if all bytes in the sequence are alphabetic ASCII characters and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

예를 들면:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()
bytearray.**isascii**()

Return `True` if the sequence is empty or all bytes in the sequence are ASCII, `False` otherwise. ASCII bytes are in the range 0-0x7F.

버전 3.7에 추가.

bytes.**isdigit**()
bytearray.**isdigit**()

Return `True` if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, `False` otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

예를 들면:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()
bytearray.**islower**()

Return `True` if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, `False` otherwise.

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**()

Return `True` if all bytes in the sequence are ASCII whitespace and the sequence is not empty, `False` otherwise. ASCII whitespace characters are those byte values in the sequence `b' \t\n\r\x0b\f'` (space, tab, newline, carriage return, vertical tab, form feed).

bytes.**istitle**()

bytearray.**istitle**()
Return True if the sequence is ASCII titlecase and the sequence is not empty, False otherwise. See *bytes.title()* for more details on the definition of "titlecase".

예를 들면:

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.**isupper**()
bytearray.**isupper**()
Return True if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, False otherwise.

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.

str.swapcase() 와는 달리 바이너리 버전의 경우 항상 `bin.swapcase().swapcase() == bin` 이 성립합니다. 임의의 유니코드 포인트에서 일반적으로 성립하지는 않지만, ASCII에서 케이스 변환은 대칭적입니다.

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.

4.8.4 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.[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'	부동 소수점 지수 형식 (소문자).	(3)
'E'	부동 소수점 지수 형식 (대문자).	(3)
'f'	부동 소수점 십진수 형식.	(3)
'F'	부동 소수점 십진수 형식.	(3)
'g'	부동 소수점 형식. 지수가 -4보다 작거나 정밀도 보다 작지 않으면 소문자 지수형식을 사용하고, 그렇지 않으면 십진수 형식을 사용합니다.	(4)
'G'	부동 소수점 형식. 지수가 -4보다 작거나 정밀도 보다 작지 않으면 대문자 지수형식을 사용하고, 그렇지 않으면 십진수 형식을 사용합니다.	(4)
'c'	단일 바이트 (정수 또는 길이 1인 바이너리 시퀀스를 허용합니다).	
`b`	바이너리 시퀀스 (버퍼 프로토콜 을 따르거나 __bytes__() 가 있는 모든 객체).	(5)
's'	's' 는 'b' 의 별칭이고 파이썬 2/3에서만 사용되어야 합니다.	(6)
'a'	바이트열 (repr(obj).encode('ascii','backslashreplace) 를 사용하여 모든 파이썬 객체를 변환합니다).	(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

버전 3.5에 추가.

4.8.5 메모리 뷰

memoryview 객체는 파이썬 코드가 버퍼 프로토콜 을 지원하는 객체의 내부 데이터에 복사 없이 접근할 수 있게 합니다.

class memoryview(*obj*)

obj 를 참조하는 *memoryview* 를 만듭니다. *obj* 는 버퍼 프로토콜을 지원해야 합니다. 버퍼 프로토콜을 지원하는 내장 객체에는 *bytes* 와 *bytearray* 가 있습니다.

A *memoryview* 는 요소 라는 개념을 갖는데, 원래 객체 *obj* 에 의해 처리되는 원자 적 메모리 단위입니다. *bytes* 와 *bytearray* 와 같은 많은 간단한 형의 경우 요소는 하나의 바이트이지만, *array.array* 와 같은 다른 형들은 더 큰 요소를 가질 수 있습니다.

`len(view)` 는 *tolist* 의 길이와 같습니다. `view.ndim = 0` 이면 길이는 1 입니다. `view.ndim = 1` 이면 길이는 뷰에 있는 요소의 개수와 같습니다. 고차원의 경우, 길이는 뷰의 중첩된 리스트 표현의 길이와 같습니다. *itemsize* 어트리뷰트는 단일 요소의 바이트 수를 알려줍니다.

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
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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')
```

'B', 'b' 'c' 형식의 해시 가능(읽기 전용) 형의 일차원 메모리 뷰는 역시 해시 가능합니다. 해시는 `hash(m) == hash(m.tobytes())` 로 정의됩니다:

```
>>> 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: 이제 일차원 메모리 뷰를 슬라이스할 수 있습니다. 이제 형식이 'B', 'b', 'c' 인 일차원 메모리 뷰는 해시 가능합니다.

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.

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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

부동 소수점 숫자와 마찬가지로, 메모리 뷰 객체의 경우 `v is w` 일 때도 `v == w` 가 성립하지 않을 수 있습니다.

버전 3.3에서 변경: 이전 버전에서는 항목 형식과 논리 배열 구조를 무시하고 원시 메모리를 비교했습니다.

tobytes()

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.

hex()

버퍼 내의 각 바이트를 두 개의 16진수로 표현한 문자열 객체를 돌려줍니다.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

버전 3.5에 추가.

tolist()

버퍼 내의 데이터를 요소들의 리스트로 돌려줍니다.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> 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.

release()

메모리 뷰 객체에 의해 노출된 하부 버퍼를 해제합니다. 많은 객체는 뷰가 그 객체에 연결될 때 특별한 조치를 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.

이 메서드가 호출된 후, 뷰에 대한 더 이상의 연산은 *ValueError* 를 일으킵니다 (여러 번 호출 될 수 있는 *release()* 자신은 예외입니다):

```
>>> 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
```

버전 3.2에 추가.

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 -> 1D입니다.

목적 형식은 *struct* 문법의 단일 요소 네이티브 형식으로 제한됩니다. 형식 중 하나는 바이트 형식 ('B', 'b', 'c') 이어야 합니다. 결과의 바이트 길이는 원래 길이와 같아야 합니다.

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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value 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)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Cast 1D/unsigned long to 2D/unsigned long:

```
>>> 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]]
```

버전 3.3에 추가.

버전 3.5에서 변경: 바이트 형식으로 변환할 때 소스 형식이 더는 제한되지 않습니다.

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
```

버전 3.3에 추가.

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)
>>> 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
```

버전 3.3에 추가.

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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.

버전 3.3에 추가.

f_contiguous
메모리가 포트란 연속 인지를 나타내는 논리값.

버전 3.3에 추가.

contiguous
메모리가 연속 인지를 나타내는 논리값.

버전 3.3에 추가.

4.9 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.

*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.

참고로, 연산자가 아닌 버전의 *union()*, *intersection()*, *difference()*, *symmetric_difference()*, *issubset()*, *issuperset()* 메서드는 임의의 이터러블을 인자로 받아들입니다. 대조적으로, 연산자를 기반으로 하는 대응 연산들은 인자가 집합일 것을 요구합니다. 이것은 오류가 발생하기 쉬운 `set('abc') & 'cbs'` 와 같은 구성을 배제하고 더 읽기 쉬운 `set('abc').intersection('cbs')` 를 선호합니다.

*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.

참고로, `__contains__()`, *remove()*, *discard()* 메서드로 제공되는 *elem* 인자는 set 일 수 있습니다. 동등한 frozenset 검색을 지원하기 위해, *elem* 으로 임시 frozenset 을 만듭니다.

4.10 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.)

딕셔너리의 키는 거의 임의의 값입니다. 해시 가능 하지 않은 값들, 즉, 리스트, 딕셔너리 또는 다른 가변형 (객체 아이덴티티 대신 값으로 비교됩니다) 은 키로 사용할 수 없습니다. 키에 사용되는 숫자 형은 숫자 비교를 위한 일반적인 규칙을 따릅니다: 두 숫자가 같다고 비교되는 경우 (`1` 과 `1.0` 처럼) 같은 딕셔너리 항목을 인덱싱하는데 서로 교환하여 사용할 수 있습니다. (그러나 컴퓨터는 부동 소수점 숫자를 근삿값으로 저장하므로 이것들을 딕셔너리 키로 사용하는 것은 현명하지 않습니다.)

딕셔너리는 *dict* 생성자뿐만 아니라 중괄호 안에 쉼표로 구분된 `key: value` 쌍을 나열해서 만들 수 있습니다, 예를 들어: `{'jack': 4098, 'sjoerd': 4127}` 또는 `{4098: 'jack', 4127: 'sjoerd'}`.

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)

선택적 위치 인자와 (비어있을 수 있는) 키워드 인자들의 집합으로부터 초기화된 새 딕셔너리를 돌려줍니다.

위치 인자가 제공되지 않으면 빈 딕셔너리가 만들어집니다. 위치 인자가 지정되고 매핑 객체인 경우, 매핑 객체와 같은 키-값 쌍을 갖는 딕셔너리가 만들어집니다. 그렇지 않으면, 위치 인자는 이터러블 객체여야 합니다. 이터러블의 각 항목은 그 자체로 정확하게 두 개의 객체가 있는 이터러블이어야 합니다. 각 항목의 첫 번째 객체는 새 딕셔너리의 키가 되고, 두 번째 객체는 해당 값이 됩니다. 키가 두 번 이상 나타나면, 그 키의 마지막 값이 새 딕셔너리의 해당 값이 됩니다.

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.

예를 들어, 다음 예제는 모두 `{"one": 1, "two": 2, "three": 3}` 와 같은 딕셔너리를 돌려줍니다:

```
>>> 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})
>>> a == b == c == d == e
True
```

첫 번째 예제에서와같이 키워드 인자는 유효한 파이썬 식별자인 키에 대해서만 작동합니다. 그 외의 경우는 모든 유효한 키를 사용할 수 있습니다.

이것들은 딕셔너리가 지원하는 연산들입니다 (그러므로, 사용자 정의 매핑 형도 지원해야 합니다):

list(d)
Return a list of all the keys used in the dictionary *d*.

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.

dict 의 서브 클래스가 method `__missing__()` 을 정의하고 *key* 가 존재하지 않는다면, `d[key]` 연산은 키 *key* 를 인자로 하여 그 메서드를 호출합니다. 그런 다음 `d[key]` 연산은 `__missing__(key)` 호출이 반환한 값이나 일으킨 예외를 그대로 반환하거나 일으킵니다. 다른 연산이나 메서드는

__missing__() 을 호출하지 않습니다. __missing__() 이 정의되어 있지 않으면 *KeyError* 를 일으킵니다. __missing__() 은 메서드 여야 합니다; 인스턴스 변수가 될 수 없습니다:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

위의 예는 *collections.Counter* 구현 일부를 보여줍니다. 다른 __missing__ 메서드가 *collections.defaultdict* 에서 사용됩니다.

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*])
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.

get(*key*[, *default*])
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*[, *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 하는 데 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.

setdefault(*key*[, *default*])
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([*other*])
other 가 제공하는 키/값 쌍으로 사전을 갱신합니다. 기존 키는 덮어씁니다. `None` 을 돌려줍니다.

update() 는 다른 딕셔너리 객체 나 키/값 쌍(길이 2인 튜플이나 다른 이터러블)을 주는 이터레이터를 모두 받아들입니다. 키워드 인자가 지정되면, 딕셔너리는 그 키/값 쌍으로 갱신됩니다: `d.update(red=1, blue=2)`.

values()
딕셔너리 값들의 새 뷰를 돌려줍니다. 뷰 객체의 설명서 을 참조하세요.

An equality comparison between one `dict.values()` view and another will always return `False`. This also applies when comparing `dict.values()` to itself:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

Dictionaries compare equal if and only if they have the same `(key, value)` pairs (regardless of ordering). Order comparisons ('<', '<=', '>=', '>') raise *TypeError*.

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.

더 보기:

types.MappingProxyType 를 *dict* 의 읽기 전용 뷰를 만드는 데 사용할 수 있습니다.

4.10.1 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, 값, 항목에 있는 경우 `True` 를 돌려줍니다 (마지막의 경우 *x* 는 `(key, value)` 튜플이어야 합니다).

키 뷰는 항목이 고유하고 해시 가능하므로 집합과 유사합니다. 모든 값이 해시 가능해서 `(key, value)` 쌍들이 고유하고 해시 가능하다면, 항목 뷰 역시 집합과 유사합니다. (값 뷰는 항목이 일반적으로 고유하지 않기 때문에 집합과 같이 취급되지 않습니다.) 집합과 유사한 뷰의 경우 추상 베이스 클래스 *collections.abc.Set* 에 정의된 모든 연산을 사용할 수 있습니다 (예를 들어, ==, <, ^).

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'}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11 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.

  전달 된 예외를 명시적으로 다시 일으켜서는 안 됩니다 - 대신, 이 메서드가 성공적으로 완료되었으며 발생 된 예외를 막지 않겠다는 의미의 거짓을 돌려주어야 합니다. 이렇게 하면 컨텍스트 관리 코드가 *__exit__()* 메서드가 실제로 실패했는지를 쉽게 감지할 수 있습니다.

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.

파이썬의 제너레이터s들과 *contextlib.contextmanager* 데코레이터는 이 프로토콜을 구현하는 편리한 방법을 제공합니다. 제너레이터 함수가 *contextlib.contextmanager* 데코레이터로 데코레이팅 되면, 데코레이팅 되지 않은 제너레이터 함수가 만드는 이터레이터 대신에 필요한 `__enter__()` 와 `__exit__()` 메서드를 구현하는 컨텍스트 관리자를 돌려줍니다.

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.

4.12 기타 내장형

인터프리터는 여러 가지 다른 객체를 지원합니다. 이것들 대부분은 한두 가지 연산만 지원합니다.

4.12.1 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.

4.12.2 클래스와 클래스 인스턴스

여기에 대해서는 objects와 class를 참조하세요.

4.12.3 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.

4.12.4 메서드

메서드는 어트리뷰트 표기법을 사용하여 호출되는 함수입니다. 두 가지 종류가 있습니다: 내장 메서드(리스트의 `append()` 같은 것들)와 클래스 인스턴스 메서드. 내장 메서드는 이를 지원하는 형에서 설명됩니다.

인스턴스를 통해 메서드(클래스 이름 공간에 정의 된 함수)에 액세스하면, 특별한 객체인 연결된 메서드 *(bound method)* (인스턴스 메서드 *(instance method)* 라고도 부릅니다) 객체를 얻게 됩니다. 호출되면 인자 목록에 `self` 인자를 추가합니다. 연결된 메서드는 두 가지 특수한 읽기 전용 어트리뷰트를 가지고 있습니다: `m.__self__` 는 메서드가 작동하는 객체이고, `m.__func__` 는 메서드를 구현하는 함수입니다. `m(arg-1, arg-2, ..., arg-n)` 을 호출하는 것은 `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)` 를 호출하는 것과 완전히 같습니다.

함수 객체와 같이, 연결된 메서드 객체는 임의 어트리뷰트를 읽는 것을 지원합니다. 그러나 메서드 어트리뷰트는 실제로 하부 함수 객체(`meth.__func__`)에 저장되기 때문에, 연결된 메서드에 메서드 어트리뷰트를 설정하는 것은 허용되지 않습니다. 메서드 어트리뷰트를 설정하려고 하면 *AttributeError* 를 일으킵니다. 메서드 어트리뷰트를 설정하려면, 명시적으로 하부 함수 객체에 설정해야 합니다:

```
>>> 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'
```

자세한 정보는 types를 보십시오.

4.12.5 코드 객체

코드 객체는 함수 바디와 같은 "의사 컴파일된" 실행 가능한 파이썬 코드를 표현하기 위해 구현에서 사용됩니다. 전역 실행 환경에 대한 참조가 없으므로 함수 객체와 다릅니다. 코드 객체는 내장 *compile()* 함수가 돌려주고, 함수 객체들로부터 __code__ 어트리뷰트를 통해 추출할 수 있습니다. *code* 모듈도 참고하십시오.

코드 객체는 *exec()* 또는 *eval()* 내장 함수에 (소스 문자열 대신) 전달하여 실행하거나 값을 구할 수 있습니다.

자세한 정보는 types를 보십시오.

4.12.6 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'>`.

4.12.7 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.

4.12.8 Ellipsis 객체

이 객체는 일반적으로 슬라이싱에 사용됩니다 (slicings 를 참조하세요). 특별한 연산을 지원하지 않습니다. 정확하게 하나의 Ellipsis 객체가 있으며, 이름은 `Ellipsis` (내장 이름) 입니다. ``type(Ellipsis)()` 는 *Ellipsis* 싱글톤을 만듭니다.

`Ellipsis` 나 `...` 로 쓰입니다.

4.12.9 NotImplemented 객체

이 객체는 비교와 이항 연산이 지원하지 않는 형에 대한 요청을 받았을 때 돌려줍니다. 자세한 정보는 comparisons를 보십시오. 정확하게 하나의 `NotImplemented` 객체가 있습니다. `type(NotImplemented)()` 는 싱글톤 인스턴스를 만듭니다.

`NotImplemented` 로 쓰입니다.

4.12.10 논리값

논리값은 두 개의 상수 객체인 `False` 와 `True` 입니다. 이것들은 논리값을 나타내기 위해 사용됩니다(하지만 다른 값도 거짓 또는 참으로 간주 될 수 있습니다). 숫자 컨텍스트(예를 들어, 산술 연산자의 인자로 사용될 때)에서는 각각 정수 0과 1처럼 작동합니다. 내장 함수 *bool()* 은 값이 논리값으로 해석될 수 있는 경우 모든 값을 논리값으로 변환하는 데 사용할 수 있습니다(위의 논리값 검사 절을 참조하세요).

각각 `False` 과 `True` 로 쓰입니다.

4.12.11 내부 객체

여기에 관한 정보는 types를 참조하십시오. 스택 프레임 객체, 트레이스백 객체 및 슬라이스 객체에 관해 설명합니다.

4.13 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.

`object.`**`__dict__`**
: 객체의 (쓰기 가능한) 어트리뷰트를 저장하는 데 사용되는 딕셔너리나 또는 기타 매핑 객체.

`instance.`**`__class__`**
: 클래스 인스턴스가 속한 클래스.

`class.`**`__bases__`**
: 클래스 객체의 베이스 클래스들의 튜플.

`definition.`**`__name__`**
: 클래스, 함수, 메서드, 디스크립터 또는 제너레이터 인스턴스의 이름.

`definition.`**`__qualname__`**
: 클래스, 함수, 메서드, 디스크립터 또는 제너레이터 인스턴스의 정규화된 이름.

  버전 3.3에 추가.

`class.`**`__mro__`**
: 이 어트리뷰트는 메서드 결정 중에 베이스 클래스를 찾을 때 고려되는 클래스들의 튜플입니다.

`class.`**`mro`**`()`
: 이 메서드는 인스턴스의 메서드 결정 순서를 사용자 정의하기 위해 메타 클래스가 재정의할 수 있습니다. 클래스 인스턴스를 만들 때 호출되며 그 결과는 *__mro__* 에 저장됩니다.

`class.`**`__subclasses__`**`()`
: 각 클래스는 직계 서브 클래스에 대한 약한 참조의 리스트를 유지합니다. 이 메서드는 아직 살아있는 모든 참조의 리스트를 돌려줍니다. 예:

```
>>> int.__subclasses__()
[<class 'bool'>]
```

4.14 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) 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) for integer string conversion: value has 8599␣
↪digits; 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'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

버전 3.7.14에 추가.

4.14.1 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*.

4.14.2 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).

버전 3.7.14에 추가.

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.

4.14.3 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 3.11.

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.

아래 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.

except 또는 finally 절에서 예외를 일으킬 때 (또는 다시 일으킬 때), __context__ 는 자동으로 마지막으로 잡힌 예외로 설정됩니다; 이 새 예외가 처리되지 않으면, 결국 표시되는 트레이스백은 원래 예외와 최종 예외를 포함합니다.

(현재 처리 중인 예외를 다시 발생시키기 위해 raise만 사용하는 대신) 새 예외를 일으킬 때, 묵시적인 예외 컨텍스트는 암시적인 예외 상황은 명시적 원인으로 보충될 수 있는데, raise와 from을 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.

5.1 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*)
이 메서드는 *tb* 를 예외의 새 트레이스백으로 설정하고 예외 객체를 돌려줍니다. 일반적으로 다음과 같은 예외 처리 코드에서 사용됩니다:

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

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.

5.2 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.)

exception EOFError
input() 함수가 데이터를 읽지 못한 상태에서 EOF (end-of-file) 조건을 만날 때 발생합니다. (주의하세요: `io.IOBase.read()` 와 *io.IOBase.readline()* 메서드는 EOF를 만날 때 빈 문자열을 돌려줍니다.)

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 임포트하려는 이름을 찾을 수 없을 때도 발생합니다.

`name`과 `path` 어트리뷰트는 생성자에 키워드 전용 인자를 사용하여 설정할 수 있습니다. 설정된 경우, 각각 임포트하려고 시도한 모듈의 이름과 예외를 유발한 파일의 경로를 나타냅니다.

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.

버전 3.6에 추가.

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.

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.

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.

참고: NotImplementedError 와 NotImplemented 는 비슷한 이름과 목적이 있습니다만, 바꿔쓸 수 없습니다. 언제 사용하는지에 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, 서브클래싱할 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
운영 체제에서 제공하는 해당 에러 메시지. POSIX에서는 C 함수 perror() 로, 윈도우에서는 FormatMessage() 로 포맷합니다.

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*, mmap.error 가 *OSError* 로 병합되었고, 생성자는 서브 클래스를 반환할 수 있습니다.

버전 3.4에서 변경: *filename* 속성은 이제 파일 시스템 인코딩으로 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 OverflowError는 요구되는 범위를 벗어난 정수의 경우도 발생합니다. C에서 부동 소수점 예외 처리의 표준화가 부족하므로, 대부분의 부동 소수점 연산은 검사되지 않습니다.

exception RecursionError
이 예외는 *RuntimeError* 에서 파생됩니다. 인터프리터가 최대 재귀 깊이 (*sys.getrecursionlimit()* 참조)가 초과하였음을 감지할 때 발생합니다.

버전 3.5에 추가: 이전에는 평범한 *RuntimeError* 가 발생했습니다.

exception ReferenceError
이 예외는 *weakref.proxy()* 함수가 만든 약한 참조 프락시가 이미 가비지 수집된 참조 대상의 어트

리뷰트를 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.

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.

버전 3.5에 추가.

exception SyntaxError
파서가 문법 오류를 만날 때 발생합니다. `import` 문에서, 내장 함수 *exec()* 나 *eval()* 호출에서, 초기 스크립트나 (대화형으로) 표준 입력을 읽을 때 발생할 수 있습니다.

세부 사항을 쉽게 확인할 수 있도록, 이 클래스의 인스턴스에는 `filename`, `lineno`, `offset` 및 `text` 어트리뷰트가 있습니다. 예외 인스턴스의 *str()*은 메시지만 돌려줍니다.

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.

이것을 파이썬 인터프리터의 저자 또는 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
이 예외는 *sys.exit()* 함수가 일으킵니다. *Exception*을 잡는 코드에 의해 우연히 잡히지 않도록, *Exception* 대신에 *BaseException* 을 상속합니다. 이렇게 하면 예외가 올바르게 전파되어 인터프리터가 종료됩니다. 처리되지 않으면, 파이썬 인터프리터가 종료됩니다; 스택 트레이스백은 인쇄되지 않습니다. 생성자는 *sys.exit()* 에 전달된 것과 같은 선택적 인자를 받아들입니다. 값이 정수이면

시스템 종료 상태를 지정합니다 (C의 exit() 함수에 전달됩니다); None 이면 종료 상태는 0입니다; 다른 형(가령 문자열)이면 객체의 값이 인쇄되고 종료 상태는 1입니다.

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.

end
object 에 있는 마지막으로 잘못된 데이터의 바로 다음 인덱스입니다.

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.

5.2.1 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. errno ENOTDIR 에 해당합니다.

exception PermissionError
적절한 접근권 (가령 파일 시스템 권한) 없이 연산을 실행하려고 할 때 발생합니다. errno EACCES 와 EPERM 에 해당합니다.

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.

버전 3.3에 추가: 위의 모든 *OSError* 서브 클래스가 추가되었습니다.

더 보기:

PEP 3151 - OS 및 IO 예외 계층 구조 재작업

5.3 경고

다음 예외는 경고 범주로 사용됩니다; 자세한 정보는 *Warning Categories* 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.

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.

exception SyntaxWarning
모호한 문법에 대한 경고의 베이스 클래스입니다.

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.

exception UnicodeWarning
유니코드와 관련된 경고의 베이스 클래스입니다.

exception BytesWarning
bytes 및 *bytearray* 와 관련된 경고의 베이스 클래스입니다.

exception ResourceWarning
자원 사용과 관련된 경고의 베이스 클래스입니다. 기본 경고 필터에 의해 무시됩니다.

버전 3.2에 추가.

5.4 예외 계층 구조

내장 예외의 클래스 계층 구조는 다음과 같습니다:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

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**
인쇄 가능한 것으로 간주하는 ASCII 문자의 문자열. *digits*, *ascii_letters*, *punctuation*, *whitespace* 의 조합입니다.

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.

튜플의 값은 개념적으로 리터럴 텍스트와 그 뒤를 따르는 하나의 치환 필드의 범위를 나타냅니다. 리터럴 텍스트가 없는 경우 (두 개의 치환 필드가 연속적으로 나타나는 경우 발생할 수 있습니다), *literal_text* 는 길이가 0인 문자열입니다. 치환 필드가 없는 경우 *field_name*, *format_spec* 및 *conversion* 값은 `None` 입니다.

get_field(*field_name*, *args*, *kwargs*)
parse() 가 반환한 *field_name* 을 (위를 보세요) 포맷될 객체로 변환합니다. 튜플 (obj, used_key) 를 반환합니다. 기본 버전은 "0[name]" 이나 "label.title"과 같이 **PEP 3101** 에 정의된 형식의 문자열을 받아들입니다. *args* 와 *kwargs* 는 *vformat()* 에 전달된 것과 같습니다. 반환 값 *used_key* 는 *get_value()* 의 *key* 매개 변수와 같은 의미가 있습니다.

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

str.format() 메서드와 *Formatter* 클래스는 포맷 문자열에 대해서 같은 문법을 공유합니다 (*Formatter* 의 경우, 서브 클래스는 그들 자신의 포맷 문자열 문법을 정의 할 수 있습니다). 문법은 포맷 문자열 리터럴 과 관련 있지만, 차이점이 있습니다.

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        ::=  <described in the next section>
```

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. 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 3.1에서 변경: 위치 인자 지정자는 *str.format()* 에서 생략 할 수 있습니다. 그래서, '{} {}'.format(a, b) 는 '{0} {1}'.format(a, b) 과 동등합니다.

버전 3.4에서 변경: 위치 인자 지정자는 *Formatter*에서 생략 할 수 있습니다.

몇 가지 간단한 포맷 문자열 예제:

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional␣
↪argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

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}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
"More {!a}"                       # Calls ascii() on the argument first
```

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

"포맷 명세" 는 포맷 문자열에 포함된 치환 필드 내에서 개별 값의 표시 방법을 정의하는 데 사용됩니다(포맷 문자열 문법 과 f-strings을 보세요). 이것들은 내장 *format()* 함수에 직접 전달될 수도 있습니다. 각 포맷 가능한 형은 포맷 명세를 해석하는 방법을 정의 할 수 있습니다.

대부분의 내장형은 포맷 명세에 대해 다음 옵션을 구현하지만, 일부 포맷 옵션은 숫자 형에서만 지원됩니다.

A general convention is that an empty format specification produces the same result as if you had called *str()* on the value. A non-empty format specification typically modifies the result.

표준 포맷 지정자 의 일반적인 형식은 다음과 같습니다:

```
format_spec      ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill             ::=  <any character>
align            ::=  "<" | ">" | "=" | "^"
sign             ::=  "+" | "-" | " "
width            ::=  digit+
grouping_option  ::=  "_" | ","
precision        ::=  digit+
type             ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s
```

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 삽입 할 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 숫자 형에게만 유효합니다. 이것은 필드 너비 바로 앞에 '0' 이 있으면 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.

'#' 옵션은 변환에 "대안 형식" 이 사용되도록 만듭니다. 대안 형식은 형별로 다르게 정의됩니다. 이 옵션은 정수, 실수, 복소수와 Decimal 형에게만 유효합니다. 정수의 경우, 이진수, 8진수 또는 16진수 출력이 사용될 때, 이 옵션은 출력값에 각각 접두사 '0b', '0o' 또는 '0x' 를 추가합니다. 실수, 복소수 및 Decimal의 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.

',' 옵션은 천 단위 구분 기호에 쉼표를 사용하도록 알립니다. 로케일을 고려하는 구분자의 경우, 대신 'n' 정수 표시 유형을 사용하십시오.

버전 3.1에서 변경: ',' 옵션을 추가했습니다 (**PEP 378** 도 보세요).

'_' 옵션은 부동 소수점 표시 유형 및 정수 표시 유형 'd' 에 대해 천 단위 구분 기호에 밑줄을 사용하도록 알립니다. 정수 표시 유형 'b', 'o', 'x' 및 'X' 의 경우 밑줄이 4자리마다 삽입됩니다. 다른 표시 유형의 경우, 이 옵션을 지정하면 에러가 발생합니다.

버전 3.6에서 변경: '_' 옵션을 추가했습니다 (**PEP 515** 도 보세요).

width is a decimal integer defining the minimum total field width, including any prefixes, separators, and other formatting characters. If not specified, then the field width will be determined by the content.

명시적 정렬이 주어지지 않을 때, *width* 필드 앞에 '0' 문자를 붙이면 숫자 형에 대해 부호를 고려하는 0 채움을 사용할 수 있습니다. 이것은 '0' 의 *fill* 문자와 '=' 의 *alignment* 유형을 갖는 것과 동등합니다.

precision 는 'f' 및 'F' 로 포맷된 부동 소수점 값의 소수점 이하 또는 'g' 또는 'G' 로 포맷된 부동 소수점 값의 소수점 앞, 뒤로 표시할 숫자의 개수를 나타내는 십진수입니다. 숫자가 아닌 유형의 경우 필드는 최대 필드 크기를 나타냅니다 - 즉, 필드 내용에서 몇 개의 문자가 사용되는지 나타냅니다. 정숫값에는 *precision* 이 허용되지 않습니다.

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.
'n'	숫자. 이는 현재 로케일 설정을 사용하여 적절한 숫자 구분 문자를 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.

부동 소수점 및 Decimal 값에 사용할 수 있는 표시 유형은 다음과 같습니다:

유형	의미
'e'	지수 표기법. 지수를 나타내는 문자 'e'를 사용하여 과학 표기법으로 숫자를 인쇄합니다. 기본 정밀도는 6 입니다.
'E'	지수 표기법. 구분 문자로 대문자 'E'를 사용한다는 것을 제외하고 'e' 와 같습니다.
'f'	고정 소수점 표기법. 숫자를 고정 소수점 숫자로 표시합니다. 기본 정밀도는 6 입니다.
'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. The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision p-1 would have exponent exp. Then if -4 <= exp < p, the number is formatted with presentation type 'f' and precision p-1-exp. Otherwise, the number is formatted with presentation type 'e' and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it, unless the '#' option is used. 양과 음의 무한대, 양과 음의 0, nans는 정밀도와 무관하게 각각 inf, -inf, 0, -0, nan 으로 포맷됩니다. 0 의 정밀도는 1 의 정밀도로 처리됩니다. 기본 정밀도는 6 입니다.
'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 적절한 숫자 구분 문자를 삽입한다는 점을 제외하면 'g' 와 같습니다.
'%'	백분율. 숫자에 100을 곱해서 고정 ('f') 형식으로 표시한 다음 백분율 기호를 붙입니다.
없음	고정 소수점 표기법이 선택될 때, 소수점 이하로 적어도 하나의 자리가 있다는 점을 제외하면 'g' 와 비슷합니다. 기본 정밀도는 특정 값을 나타내는 데 필요한 만큼 높습니다. 전체적인 효과는 *str()* 의 출력을 다른 포맷 수정자에 의해 변경된 것처럼 만드는 것입니다.

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+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'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() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

텍스트 정렬과 너비 지정:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

`%+f`, `%-f`, `% f` 대체와 부호 지정:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

`%x`, `%o` 대체와 다른 진법으로 값 변환:

```
>>> # format also supports binary numbers
>>> "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'
```

쉼표를 천 단위 구분자로 사용:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

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 템플릿 문자열

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 제공 할 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.

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):
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 합니다. 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 정의 할 수 있습니다.

  버전 3.7에 추가.

- *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.

  버전 3.2에 추가.

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.

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 re — Regular expression operations

Source code: Lib/re.py

This module provides regular expression matching operations similar to those found in Perl.

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 byte 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.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'r'`. So `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and methods on *compiled regular expressions*. The functions are shortcuts that don't require you to compile a regex object first, but miss some fine-tuning parameters.

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.

6.2.1 Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if *A* and *B* are both regular expressions, then *AB* is also a regular expression. In general, if a string *p* matches *A* and another string *q* matches *B*, the string *pq* will match AB. This holds unless *A* or *B* contain low precedence operations; boundary conditions between *A* and *B*; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book [Frie09], or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the regex-howto.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like `'A'`, `'a'`, or `'0'`, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `last` matches the string `'last'`. (In the rest of this section, we'll write RE's in `this special style`, usually without quotes, and strings to be matched `'in single quotes'`.)

Some characters, like `'|'` or `'('`, are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

Repetition qualifiers (`*`, `+`, `?`, `{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.

The special characters are:

. (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.

^ (Caret.) Matches the start of the string, and in *MULTILINE* mode also matches immediately after each newline.

$ Matches the end of the string or just before the newline at the end of the string, and in *MULTILINE* mode also matches before a newline. `foo` matches both 'foo' and 'foobar', while the regular expression `foo$` matches only 'foo'. More interestingly, searching for `foo.$` in `'foo1\nfoo2\n'` matches 'foo2' normally, but 'foo1' in *MULTILINE* mode; searching for a single `$` in `'foo\n'` will find two (empty) matches: one just before the newline, and one at the end of the string.

***** Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. `ab*` will match 'a', 'ab', or 'a' followed by any number of 'b's.

+ Causes the resulting RE to match 1 or more repetitions of the preceding RE. `ab+` will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

? Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. `ab?` will match either 'a' or 'ab'.

***?, +?, ??** The `'*'`, `'+'`, and `'?'` qualifiers 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 qualifier 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>'`.

{m} Specifies that exactly *m* copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, `a{6}` will match exactly six `'a'` characters, but not five.

{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. For example, `a{3,5}` will match from 3 to 5 `'a'` characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, `a{4,}b` will match `'aaaab'` or a thousand `'a'` characters followed by a `'b'`, but not `'aaab'`. The comma may not be omitted or the modifier would be confused with the previously described form.

{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 qualifier. 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.

**** Either escapes special characters (permitting you to match characters like `'*'`, `'?'`, and so forth), or signals a special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

[] Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. `[amk]` will match `'a'`, `'m'`, or `'k'`.
- Ranges of characters can be indicated by giving two characters and separating them by a `'-'`, for example `[a-z]` will match any lowercase ASCII letter, `[0-5][0-9]` will match all the two-digits numbers from `00` to `59`, and `[0-9A-Fa-f]` will match any hexadecimal digit. If `-` is escaped (e.g. `[a\-z]`) or if it's placed as the first or last character (e.g. `[-a]` or `[a-]`), it will match a literal `'-'`.
- Special characters lose their special meaning inside sets. For example, `[(+*)]` will match any of the literal characters `'('`, `'+'`, `'*'`, or `')'`.
- Character classes such as `\w` or `\S` (defined below) are also accepted inside a set, although the characters they match depends on whether *ASCII* or *LOCALE* mode is in force.
- Characters that are not within a range can be matched by *complementing* the set. If the first character of the set is `'^'`, all the characters that are *not* in the set will be matched. For example, `[^5]` will match any character except `'5'`, and `[^^]` will match any character except `'^'`. `^` has no special meaning if it's not the first character in the set.
- 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 both match a parenthesis.
- Support of nested sets and set operations as in Unicode Technical Standard #18 might be added in the future. This would change the syntax, so to facilitate this change a *FutureWarning* will be raised in ambiguous cases for the time being. That includes sets starting with a literal `'['` or containing literal character sequences `'--'`, `'&&'`, `'~~'`, and `'||'`. To avoid a warning escape them with a backslash.

버전 3.7에서 변경: *FutureWarning* is raised if a character set contains constructs that will change semantically in the future.

| `A|B`, where *A* and *B* can be arbitrary REs, creates a regular expression that will match either *A* or *B*. An arbitrary number of REs can be separated by the `'|'` in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by `'|'` are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once *A* matches, *B* will not be tested further, even if it would produce a longer overall match. In other words, the `'|'` operator is never greedy. To match a literal `'|'`, use `\|`, or enclose it inside a character class, as in `[|]`.

(...) Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `\number` special sequence, described below. To match the literals `'('` or `')'`, use `\(` or `\)`, or enclose them inside a character class: `[(]`, `[)]`.

(?...) This is an extension notation (a `'?'` following a `'('` is not meaningful otherwise). The first character after the `'?'` determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; `(?P<name>...)` is the only exception to this rule. Following are the currently supported extensions.

(?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: *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), and *re.X* (verbose), for the entire regular expression. (The flags are described in *Module Contents*.) 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.

(?:...) A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group *cannot* be retrieved after performing a match or referenced later in the pattern.

(?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: *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), and *re.X* (verbose), for the part of the expression.

(The flags are described in *Module Contents*.)

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 byte pattern `(?L:...)` switches to locale depending 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.

버전 3.6에 추가.

버전 3.7에서 변경: The letters `'a'`, `'L'` and `'u'` also can be used in a group.

(?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 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.

Named groups can be referenced in three contexts. If the pattern is `(?P<quote>['"]).*?(?P=quote)` (i.e. matching a string quoted with either single or double quotes):

Context of reference to group "quote"	Ways to reference it
in the same pattern itself	• `(?P=quote)` (as shown) • `\1`
when processing match object *m*	• `m.group('quote')` • `m.end('quote')` (etc.)
in a string passed to the *repl* argument of `re.sub()`	• `\g<quote>` • `\g<1>` • `\1`

(?P=name) A backreference to a named group; it matches whatever text was matched by the earlier group named *name*.

(?#...) A comment; the contents of the parentheses are simply ignored.

(?=...) Matches if `...` matches next, but doesn't consume any of the string. This is called a *lookahead assertion*. For example, `Isaac (?=Asimov)` will match `'Isaac '` only if it's followed by `'Asimov'`.

(?!...) Matches if `...` doesn't match next. This is a *negative lookahead assertion*. For example, `Isaac (?!Asimov)` will match `'Isaac '` only if it's *not* followed by `'Asimov'`.

(?<=...) Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. `(?<=abc)def` will find a match in `'abcdef'`, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that `abc` or `a|b` are allowed, but `a*` and `a{3,4}` are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the *`search()`* function rather than the *`match()`* function:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

버전 3.5에서 변경: Added support for group references of fixed length.

(?<!...) Matches if the current position in the string is not preceded by a match for `...`. This is called a *negative lookbehind assertion*. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?(id/name)yes-pattern|no-pattern) Will try to match with `yes-pattern` if the group with given *id* or *name* exists, and with `no-pattern` if it doesn't. `no-pattern` is optional and can be omitted. For example, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` is a poor email matching pattern, which will match with `'<user@host.com>'` as well as `'user@host.com'`, but not with `'<user@host.com'` nor `'user@host.com>'`.

The special sequences consist of `'\'` and a character from the list below. If the ordinary character is not an ASCII digit or an ASCII letter, then the resulting RE will match the second character. For example, `\$` matches the character `'$'`.

\number Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+) \1` matches `'the the'` or `'55 55'`, but not `'thethe'` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the `'['` and `']'` of a character class, all numeric escapes are treated as characters.

\A Matches only at the start of the string.

\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/end of the string. This means that `r'\bfoo\b'` matches `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` but not `'foobar'` or `'foo3'`.

By default Unicode alphanumerics are the ones used in Unicode patterns, but this can be changed by using the *ASCII* flag. Word boundaries are determined by the current locale if the *LOCALE* flag is used. 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'py\B'` matches `'python'`, `'py3'`, `'py2'`, but not `'py'`, `'py.'`, or `'py!'`. `\B` is just the opposite of `\b`, so word characters in Unicode 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.

\d

For Unicode (str) patterns: Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes `[0-9]`, and also many other digit characters. If the *ASCII* flag is used only `[0-9]` is matched.

For 8-bit (bytes) patterns: Matches any decimal digit; this is equivalent to `[0-9]`.

\D Matches any character which is not a decimal digit. This is the opposite of `\d`. If the *ASCII* flag is used this becomes the equivalent of `[^0-9]`.

\s

For Unicode (str) patterns: Matches Unicode whitespace characters (which includes `[ \t\n\r\f\v]`, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages). If the *ASCII* flag is used, only `[ \t\n\r\f\v]` is matched.

For 8-bit (bytes) patterns: Matches characters considered whitespace in the ASCII character set; this is equivalent to `[ \t\n\r\f\v]`.

\S Matches any character which is not a whitespace character. This is the opposite of `\s`. If the *ASCII* flag is used this becomes the equivalent of `[^ \t\n\r\f\v]`.

\w

For Unicode (str) patterns: Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the *ASCII* flag is used, only `[a-zA-Z0-9_]` is matched.

For 8-bit (bytes) patterns: 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`. If the *ASCII* flag is used this becomes the equivalent of `[^a-zA-Z0-9_]`. If the *LOCALE* flag is used, matches characters which are neither alphanumeric in the current locale nor the underscore.

\Z Matches only at the end of the string.

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

```
\a      \b      \f      \n
\r      \t      \u      \U
\v      \x      \\
```

(Note that `\b` is used to represent word boundaries, and means "backspace" only inside character classes.)

`'\u'` and `'\U'` escape sequences are only recognized in Unicode patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Octal escapes are included in a limited form. If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

버전 3.3에서 변경: The `'\u'` and `'\U'` escape sequences have been added.

버전 3.6에서 변경: Unknown escapes consisting of `'\'` and an ASCII letter now are errors.

6.2.2 Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

버전 3.6에서 변경: Flag constants are now instances of `RegexFlag`, which is a subclass of *enum.IntFlag*.

`re.`**compile**(*pattern*, *flags=0*)

Compile a regular expression pattern into a *regular expression object*, which can be used for matching using its *match()*, *search()* and other methods, described below.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the following variables, combined using bitwise OR (the `|` operator).

The sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```
result = re.match(pattern, string)
```

but using *re.compile()* and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

참고: The compiled versions of the most recent patterns passed to *re.compile()* and the module-level matching functions are cached, so programs that use only a few regular expressions at a time needn't worry about compiling regular expressions.

---

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 patterns, and is ignored for byte patterns. Corresponds to the inline flag (?a).

Note that for backward compatibility, the re.U flag still exists (as well as its synonym re.UNICODE and its embedded counterpart (?u)), but these are redundant in Python 3 since matches are Unicode by default for strings (and Unicode matching isn't allowed for bytes).

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 *re.ASCII* flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the *re.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. The use of this flag is discouraged as the locale mechanism is very unreliable, it only handles one "culture" at a time, and it only works with 8-bit locales. Unicode matching is already enabled by default in Python 3 for Unicode (str) patterns, and it is able to handle different locales/languages. Corresponds to the inline flag (?L).

버전 3.6에서 변경: *re.LOCALE* can be used only with bytes patterns and is not compatible with *re.ASCII*.

버전 3.7에서 변경: Compiled regular expression objects with the *re.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.**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.**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<...>`. 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.

This means that the two following regular expression objects that match a decimal number are functionally equal:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Corresponds to the inline flag `(?x)`.

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 object*. 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.

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 object*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in *MULTILINE* mode, *re.match()* will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in *string*, use *search()* instead (see also *search() vs. match()*).

re.**fullmatch**(*pattern*, *string*, *flags=0*)
If the whole *string* matches the regular expression *pattern*, return a corresponding *match object*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

버전 3.4에 추가.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)
Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.

```
>>> 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.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

That way, separator components are always found at the same relative indices within the result list.

Empty matches for the pattern split the string only when not adjacent to a previous 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', '...', '', '', '']
```

버전 3.1에서 변경: Added the optional flags argument.

버전 3.7에서 변경: Added support of splitting on a pattern that could match an empty string.

re.**findall**(*pattern*, *string*, *flags=0*)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings. The *string* is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.

버전 3.7에서 변경: Non-empty matches can now start just after a previous empty match.

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.

버전 3.7에서 변경: Non-empty matches can now start just after a previous empty match.

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes of ASCII letters are reserved for future use and treated as errors. Other unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
>>> 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 object* 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 object*.

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. Empty matches for the pattern are replaced only when not adjacent to a previous empty match, so `sub('x*', '-', 'abxd')` returns `'-a-b--d-'`.

In string-type *repl* arguments, in addition to the character escapes and backreferences described above, `\g<name>` will use the substring matched by the group named `name`, as defined by the `(?P<name>...)` syntax. `\g<number>` uses the corresponding group number; `\g<2>` is therefore equivalent to `\2`, but isn't ambiguous in a replacement such as `\g<2>0`. `\20` would be interpreted as a reference to group 20, not a reference to group

2 followed by the literal character `'0'`. The backreference `\g<0>` substitutes in the entire substring matched by the RE.

버전 3.1에서 변경: Added the optional flags argument.

버전 3.5에서 변경: Unmatched groups are replaced with an empty string.

버전 3.6에서 변경: Unknown escapes in *pattern* consisting of `'\'` and an ASCII letter now are errors.

버전 3.7에서 변경: Unknown escapes in *repl* consisting of `'\'` and an ASCII letter now are errors.

버전 3.7에서 변경: Empty matches for the pattern are replaced when adjacent to a previous non-empty match.

`re.`**`subn`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Perform the same operation as *sub()*, but return a tuple `(new_string, number_of_subs_made)`.

버전 3.1에서 변경: Added the optional flags argument.

버전 3.5에서 변경: Unmatched groups are replaced with an empty string.

`re.`**`escape`**(*pattern*)

Escape special characters in *pattern*. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```
>>> print(re.escape('http://www.python.org'))
http://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))))
/|\-|\+|\*\*|\*
```

This function must not be used for the replacement string in *sub()* and *subn()*, only backslashes should be escaped. For example:

```
>>> 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: The `'_'` character is no longer escaped.

버전 3.7에서 변경: Only characters that can have special meaning in a regular expression are escaped. As a result, `'!'`, `'"'`, `'%'`, `"'"`, `','`, `'/'`, `':'`, `';'`, `'<'`, `'='`, `'>'`, `'@'`, and `"`` "` are no longer escaped.

`re.`**`purge`**()

Clear the regular expression cache.

`exception` `re.`**`error`**(*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 error instance has the following additional attributes:

`msg`

The unformatted error message.

`pattern`

The regular expression pattern.

pos
The index in *pattern* where compilation failed (may be `None`).

lineno
The line corresponding to *pos* (may be `None`).

colno
The column corresponding to *pos* (may be `None`).

버전 3.5에서 변경: Added additional attributes.

6.2.3 Regular Expression Objects

Compiled regular expression objects support the following methods and attributes:

`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 object*. 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 optional second parameter *pos* gives an index in the string where the search is to start; it defaults to `0`. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is *endpos* characters long, so only the characters from *pos* to `endpos - 1` will be searched for a match. If *endpos* is less than *pos*, no match will be found; otherwise, if *rx* is a compiled regular expression object, `rx.search(string, 0, 50)` is equivalent to `rx.search(string[:50], 0)`.

```
>>> 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 object*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the `search()` method.

```
>>> 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'>
```

If you want to locate a match anywhere in *string*, use `search()` instead (see also *search() vs. match()*).

`Pattern.`**`fullmatch`**`(`*string*`[, `*pos*`[, `*endpos*`]])`
If the whole *string* matches this regular expression, return a corresponding *match object*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the `search()` method.

```
>>> 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.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

버전 3.4에 추가.

Pattern.**split**(*string*, *maxsplit=0*)
Identical to the *split()* function, using the compiled pattern.

Pattern.**findall**(*string*[, *pos*[, *endpos*]])
Similar to the *findall()* function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for *search()*.

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])
Similar to the *finditer()* function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for *search()*.

Pattern.**sub**(*repl*, *string*, *count=0*)
Identical to the *sub()* function, using the compiled pattern.

Pattern.**subn**(*repl*, *string*, *count=0*)
Identical to the *subn()* function, using the compiled pattern.

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**
The number of capturing groups in the pattern.

Pattern.**groupindex**
A dictionary mapping any symbolic group names defined by `(?P<id>)` to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

Pattern.**pattern**
The pattern string from which the pattern object was compiled.

버전 3.7에서 변경: Added support of *copy.copy()* and *copy.deepcopy()*. Compiled regular expression objects are considered atomic.

6.2.4 Match Objects

Match objects always have a boolean value of `True`. Since *match()* and *search()* return `None` when there is no match, you can test whether there was a match with a simple `if` statement:

```
match = re.search(pattern, string)
if match:
    process(match)
```

Match objects support the following methods and attributes:

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.

버전 3.5에서 변경: Unmatched groups are replaced with an empty string.

Match.**group**([*group1*, ...])
Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an *IndexError* exception

is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```
>>> 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')
```

If the regular expression uses the `(?P<name>...)` syntax, the *groupN* arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an *IndexError* exception is raised.

A moderately complicated example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

`Match.__getitem__`(*g*)

This is identical to `m.group(g)`. This allows easier access to an individual group from a match:

```
>>> 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'
```

버전 3.6에 추가.

`Match.groups`(*default=None*)

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to `None`.

For example:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to `None` unless the *default* argument is given:

```
>>> 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*`)`

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to `None`. For example:

```
>>> 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*]`)`

Return the indices of the start and end of the substring matched by *group*; *group* defaults to zero (meaning the whole matched substring). Return `-1` if *group* exists but did not contribute to the match. For a match object *m*, and a group *g* that did contribute to the match, the substring matched by group *g* (equivalent to `m.group(g)`) is

```
m.string[m.start(g):m.end(g)]
```

Note that `m.start(group)` will equal `m.end(group)` if *group* matched a null string. For example, after `m = re.search('b(c?)', 'cba')`, `m.start(0)` is 1, `m.end(0)` is 2, `m.start(1)` and `m.end(1)` are both 2, and `m.start(2)` raises an *IndexError* exception.

An example that will remove *remove_this* from email addresses:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

`Match.`**`span`**`(`[*group*]`)`

For a match *m*, return the 2-tuple `(m.start(group), m.end(group))`. Note that if *group* did not contribute to the match, this is `(-1, -1)`. *group* defaults to zero, the entire match.

`Match.`**`pos`**

The value of *pos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string at which the RE engine started looking for a match.

`Match.`**`endpos`**

The value of *endpos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string beyond which the RE engine will not go.

`Match.`**`lastindex`**

The integer index of the last matched capturing group, or `None` if no group was matched at all. For example, the expressions `(a)b`, `((a)(b))`, and `((ab))` will have `lastindex == 1` if applied to the string `'ab'`, while the expression `(a)(b)` will have `lastindex == 2`, if applied to the same string.

Match.**lastgroup**
The name of the last matched capturing group, or None if the group didn't have a name, or if no group was matched at all.

Match.**re**
The *regular expression object* whose *match()* or *search()* method produced this match instance.

Match.**string**
The string passed to *match()* or *search()*.

버전 3.7에서 변경: Added support of *copy.copy()* and *copy.deepcopy()*. Match objects are considered atomic.

6.2.5 Regular Expression Examples

Checking for a Pair

In this example, we'll use the following helper function to display match objects a little more gracefully:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Suppose you are writing a poker program where a player's hand is represented as a 5-character string with each character representing a card, "a" for ace, "k" for king, "q" for queen, "j" for jack, "t" for 10, and "2" through "9" representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```
>>> 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=()>"
```

That last hand, "727ak", contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```
>>> 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',)>"
```

To find out what card the pair consists of, one could use the *group()* method of the match object in the following manner:

```
>>> 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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  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'
```

Simulating scanf()

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	Regular Expression
`%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]+`

To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python offers two different primitive operations based on regular expressions: *`re.match()`* checks for a match only at the beginning of the string, while *`re.search()`* checks for a match anywhere in the string (this is what Perl does by default).

For example:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
```

Regular expressions beginning with `'^'` can be used with *`search()`* to restrict the match at the beginning of the string:

```
>>> 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'>
```

Note however that in *MULTILINE* mode *match()* only matches at the beginning of the string, whereas using *search()* with a regular expression beginning with `'^'` will match at the beginning of each line.

```
>>> 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'>
```

Making a Phonebook

split() splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax:

```
>>> 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"""
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```
>>> 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']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the `maxsplit` parameter of *split()* because the address has spaces, our splitting pattern, in it:

```
>>> [re.split(":? ", entry, 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']]
```

The `:?` pattern matches the colon after the last name, so that it does not occur in the result list. With a `maxsplit` of `4`, we could separate the house number from the street name:

```
>>> [re.split(":? ", entry, 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']]
```

Text Munging

sub() replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using *sub()* with a function to "munge" text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```
>>> 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.'
```

Finding all Adverbs

findall() matches *all* occurrences of a pattern, not just the first one as *search()* does. For example, if a writer wanted to find all of the adverbs in some text, they might use *findall()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

Finding all Adverbs and their Positions

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", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Raw String Notation

Raw string notation (`r"text"`) keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
>>> 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 '>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use `"\\\\"`, making the following lines of code functionally identical:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

Writing a Tokenizer

A tokenizer or scanner analyzes a string to categorize groups of characters. This is a useful first step in writing a compiler or interpreter.

The text categories are specified with regular expressions. The technique is to combine those into a single master regular expression and to loop over successive matches:

```
import collections
import re

Token = collections.namedtuple('Token', ['type', 'value', 'line', 'column'])

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
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'''

for token in tokenize(statements):
    print(token)
```

The tokenizer produces the following output:

```
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)
```

6.3 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 차이 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 이상이면, 이 항목은 "흔한"

것으로 표시되고 시퀀스 일치를 위해 정크로 처리됩니다. 이 경험적 방법은 *SequenceMatcher*를 만들 때 autojunk 인자를 False로 설정하여 끌 수 있습니다.

버전 3.2에 추가: *autojunk* 매개 변수.

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 탭 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 8입니다.

*wrapcolumn*는 줄이 자동 줄 넘김 되는 열 번호를 지정하는 선택적 키워드로, 줄을 자동 줄 넘김 하지 않는 None이 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 True로 설정하십시오, 그렇지 않으면 기본값은 전체 파일을 표시하는 False입니다. *numlines*의 기본값은 5입니다. *context*가 True 일 때, *numlines*는 차이 하이라이트를 둘러싸는 문맥 줄의 수를 제어합니다. *context*가 False면 *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).

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.

Tools/scripts/diff.py는 이 클래스의 명령 줄 프런트엔드며, 좋은 사용 예를 담고 있습니다.

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.

```
>>> 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 *difflib의 명령 줄 인터페이스*를 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 0보다 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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
[]
>>> 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 None입니다. 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.

Tools/scripts/ndiff.py는 이 함수에 대한 명령 줄 프런트엔드 입니다.

```
>>> 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.

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.

```
>>> 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.

버전 3.5에 추가.

difflib.**IS_LINE_JUNK**(*line*)

Return `True` for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in *ndiff()* in older versions.

difflib.**IS_CHARACTER_JUNK**(*ch*)

Return `True` for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in *ndiff()*.

더 보기:

Pattern Matching: The Gestalt Approach John W. Ratcliff와 D. E. Metzener의 비슷한 알고리즘에 관한 토론. 이것은 1988년 7월 Dr. Dobb's Journal에 출판되었습니다.

6.3.1 SequenceMatcher 객체

SequenceMatcher 클래스는 다음과 같은 생성자를 갖습니다:

class difflib.**SequenceMatcher**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: False`; in other words, no elements are ignored. For example, pass:

```
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 3.2에 추가: *autojunk* 매개 변수.

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.

버전 3.2에 추가: *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*, *ahi*, *blo*, *bhi*)

`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.

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: Caution: The result of a *ratio()* call may depend on the order of the arguments. For instance:

```
>>> 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
```

6.3.2 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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.

6.3.3 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 None이며, 이는 어떤 줄도 정크로 간주하지 않음을 의미합니다.

charjunk: 문자(길이 1의 문자열)를 받아들이고, 문자가 정크면 참을 반환하는 함수입니다. 기본값은 None이며, 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.

6.3.4 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.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
?           ++++ ^                         ^
+   5. Flat is better than nested.
```

6.3.5 difflib의 명령 줄 인터페이스

이 예제는 difflib를 사용하여 `diff`와 유사한 유틸리티를 만드는 방법을 보여줍니다. 이것은 파이썬 소스 배포판에 `Tools/scripts/diff.py`로 포함되어 있습니다.

```
#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4 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*, ***kwargs*)
: *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. *width*의 기본값은 `70`입니다.

  wrap() 작동 방식에 대한 자세한 내용은 *TextWrapper.wrap()* 메서드를 참조하십시오.

textwrap.**fill**(*text*, *width=70*, ***kwargs*)
: *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*, ***kwargs*)
: 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.

버전 3.4에 추가.

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.

Lines containing only whitespace are ignored in the input and normalized to a single newline character in the output.

예를 들면:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))           # prints '    hello\n      world\n    '
    print(repr(dedent(s)))   # prints 'hello\n  world\n'
```

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
```

버전 3.3에 추가.

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.

버전 3.3에 추가.

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.

Since the sentence detection algorithm relies on `string.lowercase` for the definition of "lowercase letter", and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

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) None이 아니면, 출력은 최대 *max_lines* 줄을 포함하고, *placeholder*가 출력 끝에 나타납니다.

버전 3.4에 추가.

placeholder
(기본값: ' [...]') 잘렸을 때 출력 텍스트의 끝에 표시할 문자열.

버전 3.4에 추가.

*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.

6.5 unicodedata — 유니코드 데이터베이스

이 모듈은 모든 유니코드 문자에 대한 문자 속성을 정의하는 유니코드 문자 데이터베이스(UCD – Unicode Character Database)에 대한 액세스를 제공합니다. 이 데이터베이스에 포함된 데이터는 UCD 버전 11.0.0으로 컴파일됩니다.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, "Unicode Character Database". It defines the following functions:

unicodedata.**lookup**(*name*)
이름으로 문자를 조회합니다. 지정된 이름의 문자가 발견되면, 대응하는 문자를 돌려줍니다. 발견되지 않으면, *KeyError*가 발생합니다.

버전 3.3에서 변경: 이름 별칭[1] 과 명명된 시퀀스[2] 가 추가되었습니다.

unicodedata.**name**(*chr*[, *default*])
chr 문자에 할당된 이름을 문자열로 반환합니다. 이름이 정의되지 않으면, *default*가 반환되거나, 지정되지 않으면 *ValueError*가 발생합니다.

unicodedata.**decimal**(*chr*[, *default*])
chr 문자에 할당된 10진수 값을 정수로 반환합니다. 그러한 값이 정의되어 있지 않으면 *default*가 반환되거나, 지정되지 않으면 *ValueError*가 발생합니다.

unicodedata.**digit**(*chr*[, *default*])
chr 문자에 할당된 숫자(digit) 값을 정수로 반환합니다. 그러한 값이 정의되어 있지 않으면 *default*가 반환되거나, 지정되지 않으면 *ValueError*가 발생합니다.

unicodedata.**numeric**(*chr*[, *default*])
chr 문자에 할당된 수치(numeric value)를 float로 반환합니다. 그러한 값이 정의되어 있지 않으면 *default*가 반환되거나, 지정되지 않으면 *ValueError*가 발생합니다.

unicodedata.**category**(*chr*)
chr 문자에 할당된 일반 범주(general category)를 문자열로 반환합니다.

unicodedata.**bidirectional**(*chr*)
chr 문자에 할당된 양방향 클래스(bidirectional class)를 문자열로 반환합니다. 그러한 값이 정의되어 있지 않으면, 빈 문자열이 반환됩니다.

unicodedata.**combining**(*chr*)
chr 문자에 할당된 정준 결합 클래스(canonical combining class)를 정수로 반환합니다. 결합 클래스가 정의되지 않으면 0을 반환합니다.

unicodedata.**east_asian_width**(*chr*)
문자 *chr*에 할당된 동아시아 폭(east asian width)을 문자열로 반환합니다.

unicodedata.**mirrored**(*chr*)
문자 *chr*에 할당된 거울상 속성(mirrored property)을 정수로 반환합니다. 문자가 양방향 텍스트에서 "거울상" 문자로 식별되면 1을 반환하고, 그렇지 않으면 0을 반환합니다.

unicodedata.**decomposition**(*chr*)
문자 *chr*에 할당된 문자 분해 매핑(character decomposition mapping)을 문자열로 반환합니다. 그러한 매핑이 정의되어 있지 않으면 빈 문자열이 반환됩니다.

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.

[1] http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt
[2] http://www.unicode.org/Public/11.0.0/ucd/NamedSequences.txt

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.

이 두 형식 외에도, 호환 등가성을 기반으로 하는 두 가지 추가 정규화 형식이 있습니다. 유니코드에서는, 일반적으로 다른 문자와 통합되는 특정 문자가 지원됩니다. 예를 들어, U+2160 (ROMAN NUMERAL ONE) 은 U+0049 (LATIN CAPITAL LETTER I) 과 실제로 같습니다. 하지만, 기존 문자 집합(예를 들어, gb2312)과의 호환성을 위해 유니코드에서 지원됩니다.

정규화 형식 KD(NFKD)는 호환 분해를 적용합니다. 즉, 모든 호환 문자를 동등한 것으로 치환합니다. 정규화 형식 KC(NFKC)는 먼저 호환 분해를 적용한 다음, 정준 결합을 적용합니다.

두 개의 유니코드 문자열이 정규화되고, 사람이 보기에 같아 보여도, 하나가 결합한 문자를 갖고 다른 것은 그렇지 않으면, 같다고 비교되지 않을 수 있습니다.

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.

예제:

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

6.6 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.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, *`stringprep`* provides the "characteristic function", i.e. a function that returns `True` if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

`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.

6.7 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.

참고: 하부 Readline 라이브러리 API는 GNU readline 대신 `libedit` 라이브러리로 구현될 수 있습니다. macOS에서 *readline* 모듈은 실행 시간에 사용 중인 라이브러리를 감지합니다.

`libedit`의 구성 파일은 GNU readline의 구성 파일과 다릅니다. 프로그래밍 방식으로 구성 문자열을 로드하는 경우 `readline.__doc__`에서 "libedit" 텍스트를 확인하여 GNU readline과 libedit를 구별할 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
```

6.7.1 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*])
readline 초기화 파일을 실행합니다. 기본 파일 이름은 마지막으로 사용한 파일 이름입니다. 하부 라이브러리에서 `rl_read_init_file()`을 호출합니다.

6.7.2 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.

6.7.3 히스토리 파일

다음 함수는 히스토리 파일에 대해 작용합니다:

readline.**read_history_file**([*filename*])
readline 히스토리 파일을 로드하고, 히스토리 목록에 추가합니다. 기본 파일명은 ~/.history입니다. 하부 라이브러리에서 read_history()를 호출합니다.

readline.**write_history_file**([*filename*])
히스토리 목록을 readline 히스토리 파일에 저장하여, 기존 파일을 덮어씁니다. 기본 파일명은 ~/.history입니다. 하부 라이브러리에서 write_history()를 호출합니다.

readline.**append_history_file**(*nelements*[, *filename*])
히스토리의 마지막 *nelements* 항목을 파일에 추가합니다. 기본 파일명은 ~/.history입니다. 파일이 이미 존재해야 합니다. 하부 라이브러리에서 append_history()를 호출합니다. 이 함수는 파이썬이 이를 지원하는 라이브러리 버전으로 컴파일된 경우에만 존재합니다.

버전 3.5에 추가.

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.

6.7.4 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*)
readline을 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.

버전 3.6에 추가.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 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 readline이 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 `None`이면, 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.

6.7.6 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 `None`이면, 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 `None`을 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.

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.

6.7.7 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

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 rlcompleter — GNU readline을 위한 완성 함수

소스 코드: Lib/rlcompleter.py

---

rlcompleter 모듈은 유효한 파이썬 식별자와 키워드를 완성함으로써 *readline* 모듈에 적합한 완성 함수를 정의합니다.

readline 모듈을 사용할 수 있는 유닉스 플랫폼에서 이 모듈이 임포트될 때, `Completer` 클래스의 인스턴스가 자동으로 만들어지고, `complete()` 메서드가 *readline* 완성기(completer)로 설정됩니다.

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(
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.

6.8.1 Completer 객체

Completer 객체는 다음과 같은 메서드를 가집니다:

`Completer.`**`complete`**(*text*, *state*)

*text*에 대한 *state* 번째 완성을 반환합니다.

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` — Interpret bytes as packed binary data

Source code: Lib/struct.py

This module performs conversions between Python values and C structs represented as Python `bytes` objects. This can be used in handling binary data stored in files or from network connections, among other sources. It uses *Format Strings* as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

참고: By default, the result of packing a given C struct includes pad bytes in order to maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. This behavior is chosen so that the bytes of a packed struct correspond exactly to the layout in memory of the corresponding C struct. To handle platform-independent data formats or omit implicit pad bytes, use `standard` size and alignment instead of `native` size and alignment: see *Byte Order, Size, and Alignment* for details.

Several `struct` functions (and methods of `Struct`) take a *buffer* argument. This refers to objects that implement the bufferobjects and provide either a readable or read-writable buffer. The most common types used for that purpose are `bytes` and `bytearray`, but many other types that can be viewed as an array of bytes implement the buffer protocol, so that they can be read/filled without additional copying from a `bytes` object.

7.1.1 Functions and Exceptions

The module defines the following exception and functions:

exception struct.**error**
Exception raised on various occasions; argument is a string describing what is wrong.

struct.**pack**(*format*, *v1*, *v2*, ...)
Return a bytes object containing the values *v1*, *v2*, ⋯ packed according to the format string *format*. The arguments must match the values required by the format exactly.

struct.**pack_into**(*format*, *buffer*, *offset*, *v1*, *v2*, ...)
Pack the values *v1*, *v2*, ⋯ according to the format string *format* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

struct.**unpack**(*format*, *buffer*)
Unpack from the buffer *buffer* (presumably packed by `pack(format, ...)`) according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by *calcsize()*.

struct.**unpack_from**(*format*, *buffer*, *offset=0*)
Unpack from *buffer* starting at position *offset*, according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes, minus *offset*, must be at least the size required by the format, as reflected by *calcsize()*.

struct.**iter_unpack**(*format*, *buffer*)
Iteratively unpack from the buffer *buffer* according to the format string *format*. This function returns an iterator which will read equally-sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by *calcsize()*.

Each iteration yields a tuple as specified by the format string.

버전 3.4에 추가.

struct.**calcsize**(*format*)
Return the size of the struct (and hence of the bytes object produced by `pack(format, ...)`) corresponding to the format string *format*.

7.1.2 Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They are built up from *Format Characters*, which specify the type of data being packed/unpacked. In addition, there are special characters for controlling the *Byte Order, Size, and Alignment*.

Byte Order, Size, and Alignment

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

If the first character is not one of these, `'@'` is assumed.

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86 and AMD64 (x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use `sys.byteorder` to check the endianness of your system.

Native size and alignment are determined using the C compiler's `sizeof` expression. This is always combined with native byte order.

Standard size depends only on the format character; see the table in the *Format Characters* section.

Note the difference between `'@'` and `'='`: both use native byte order, but the size and alignment of the latter is standardized.

The form `'!'` is available for those poor souls who claim they can't remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of `'<'` or `'>'`.

Notes:

(1) Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.

(2) No padding is added when using non-native size and alignment, e.g. with '<', '>', '=', and '!'.

(3) To align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. See *Examples*.

Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The 'Standard size' column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	bytes of length 1	1	
b	signed char	integer	1	(1), (2)
B	unsigned char	integer	1	(2)
?	_Bool	bool	1	(1)
h	short	integer	2	(2)
H	unsigned short	integer	2	(2)
i	int	integer	4	(2)
I	unsigned int	integer	4	(2)
l	long	integer	4	(2)
L	unsigned long	integer	4	(2)
q	long long	integer	8	(2)
Q	unsigned long long	integer	8	(2)
n	ssize_t	integer		(3)
N	size_t	integer		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	bytes		
p	char[]	bytes		
P	void *	integer		(5)

버전 3.3에서 변경: Added support for the `'n'` and `'N'` formats.

버전 3.6에서 변경: Added support for the `'e'` format.

Notes:

(1) The `'?'` conversion code corresponds to the `_Bool` type defined by C99. If this type is not available, it is simulated using a `char`. In standard mode, it is always represented by one byte.

(2) When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a `__index__()` method then that method is called to convert the argument to an integer before packing.

버전 3.2에서 변경: Use of the `__index__()` method for non-integers is new in 3.2.

(3) The `'n'` and `'N'` conversion codes are only available for the native size (selected as the default or with the `'@'` byte order character). For the standard size, you can use whichever of the other integer formats fits your application.

(4) For the `'f'`, `'d'` and `'e'` conversion codes, the packed representation uses the IEEE 754 binary32, binary64 or binary16 format (for `'f'`, `'d'` or `'e'` respectively), regardless of the floating-point format used by the platform.

(5) The `'P'` format character is only available for the native byte ordering (selected as the default or with the `'@'` byte order character). The byte order character `'='` chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the `'P'` format is not available.

(6) The IEEE 754 binary16 "half precision" type was introduced in the 2008 revision of the IEEE 754 standard. It has a sign bit, a 5-bit exponent and 11-bit precision (with 10 bits explicitly stored), and can represent numbers between approximately `6.1e-05` and `6.5e+04` at full precision. This type is not widely supported by C compilers: on a typical machine, an unsigned short can be used for storage, but not for math operations. See the Wikipedia page on the half-precision floating-point format for more information.

A format character may be preceded by an integral repeat count. For example, the format string `'4h'` means exactly the same as `'hhhh'`.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the `'s'` format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters; for example, `'10s'` means a single 10-byte string, while `'10c'` means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

When packing a value `x` using one of the integer formats (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), if `x` is outside the valid range for that format then *`struct.error`* is raised.

버전 3.1 에서 변경: In 3.0, some of the integer formats wrapped out-of-range values and raised *`DeprecationWarning`* instead of *`struct.error`*.

The `'p'` format character encodes a "Pascal string", meaning a short variable-length string stored in a *fixed number of bytes*, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to *`pack()`* is too long (longer than the count minus 1), only the leading `count-1` bytes of the string are stored. If the string is shorter than `count-1`, it is padded with null bytes so that exactly count bytes in all are used. Note that for *`unpack()`*, the `'p'` format character consumes `count` bytes, but that the string returned can never contain more than 255 bytes.

For the `'?'` format character, the return value is either *`True`* or *`False`*. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be `True` when unpacking.

Examples

참고: All examples assume a native byte order, size, and alignment with a big-endian machine.

A basic example of packing/unpacking three integers:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```
>>> 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 since the padding needed to satisfy alignment requirements is different:

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
8
>>> calcsize('ic')
5
```

The following format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries:

```
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

더 보기:

Module ***array*** Packed binary storage of homogeneous data.

Module ***xdrlib*** Packing and unpacking of XDR data.

7.1.3 Classes

The *struct* module also defines the following type:

class struct.**Struct**(*format*)

Return a new Struct object which writes and reads binary data according to the format string *format*. Creating a Struct object once and calling its methods is more efficient than calling the *struct* functions with the same format since the format string only needs to be compiled once.

참고: The compiled versions of the most recent format strings passed to *Struct* and the module-level functions are cached, so programs that use only a few format strings needn't worry about reusing a single *Struct* instance.

Compiled Struct objects support the following methods and attributes:

pack(*v1*, *v2*, ...)

Identical to the *pack()* function, using the compiled format. (`len(result)` will equal *size*.)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

Identical to the *pack_into()* function, using the compiled format.

unpack(*buffer*)

Identical to the *unpack()* function, using the compiled format. The buffer's size in bytes must equal *size*.

unpack_from(*buffer*, *offset=0*)

Identical to the *unpack_from()* function, using the compiled format. The buffer's size in bytes, minus *offset*, must be at least *size*.

iter_unpack(*buffer*)

Identical to the *iter_unpack()* function, using the compiled format. The buffer's size in bytes must be a multiple of *size*.

버전 3.4에 추가.

format

The format string used to construct this Struct object.

버전 3.7에서 변경: The format string type is now *str* instead of *bytes*.

size

The calculated size of the struct (and hence of the bytes object produced by the *pack()* method) corresponding to *format*.

7.2 codecs — Codec registry and base classes

Source code: Lib/codecs.py

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are *text encodings*, which encode text to bytes, but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to use specifically with *text encodings*, or with codecs that encode to `bytes`.

The module defines the following functions for encoding and decoding with any codec:

codecs.**encode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
Encodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that encoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeEncodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

codecs.**decode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
Decodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that decoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeDecodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

The full details for each codec can also be looked up directly:

codecs.**lookup**(*encoding*)
Looks up the codec info in the Python codec registry and returns a `CodecInfo` object as defined below.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no `CodecInfo` object is found, a `LookupError` is raised. Otherwise, the `CodecInfo` object is stored in the cache and returned to the caller.

class codecs.**CodecInfo**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)
Codec details when looking up the codec registry. The constructor arguments are stored in attributes of the same name:

name
The name of the encoding.

encode
decode
The stateless encoding and decoding functions. These must be functions or methods which have the same interface as the `encode()` and `decode()` methods of Codec instances (see *Codec Interface*). The functions or methods are expected to work in a stateless mode.

incrementalencoder
incrementaldecoder
Incremental encoder and decoder classes or factory functions. These have to provide the interface defined by the base classes `IncrementalEncoder` and `IncrementalDecoder`, respectively. Incremental codecs can maintain state.

streamwriter
streamreader
Stream writer and reader classes or factory functions. These have to provide the interface defined by the base classes `StreamWriter` and `StreamReader`, respectively. Stream codecs can maintain state.

To simplify access to the various codec components, the module provides these additional functions which use *lookup()* for the codec lookup:

codecs.**getencoder**(*encoding*)
Look up the codec for the given encoding and return its encoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getdecoder**(*encoding*)
Look up the codec for the given encoding and return its decoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getincrementalencoder**(*encoding*)
Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental encoder.

codecs.**getincrementaldecoder**(*encoding*)
Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental decoder.

codecs.**getreader**(*encoding*)
Look up the codec for the given encoding and return its *StreamReader* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getwriter**(*encoding*)
Look up the codec for the given encoding and return its *StreamWriter* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

Custom codecs are made available by registering a suitable codec search function:

codecs.**register**(*search_function*)
Register a codec search function. Search functions are expected to take one argument, being the encoding name in all lower case letters, and return a *CodecInfo* object. In case a search function cannot find a given encoding, it should return `None`.

참고: Search function registration is not currently reversible, which may cause problems in some cases, such as unit testing or module reloading.

While the builtin *open()* and the associated *io* module are the recommended approach for working with encoded text files, this module provides additional utility functions and classes that allow the use of a wider range of codecs when working with binary files:

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=1*)
Open an encoded file using the given *mode* and return an instance of *StreamReaderWriter*, providing transparent encoding/decoding. The default file mode is `'r'`, meaning to open the file in read mode.

참고: Underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in *open()* function; the `'b'` is automatically added.

---

encoding specifies the encoding which is to be used for the file. Any encoding that encodes to and decodes from bytes is allowed, and the data types supported by the file methods depend on the codec used.

errors may be given to define the error handling. It defaults to `'strict'` which causes a *ValueError* to be raised in case an encoding error occurs.

buffering has the same meaning as for the built-in *open()* function. It defaults to line buffered.

codecs.**EncodedFile**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)
Return a *StreamRecoder* instance, a wrapped version of *file* which provides transparent transcoding. The original file is closed when the wrapped version is closed.

Data written to the wrapped file is decoded according to the given *data_encoding* and then written to the original file as bytes using *file_encoding*. Bytes read from the original file are decoded according to *file_encoding*, and the result is encoded using *data_encoding*.

If *file_encoding* is not given, it defaults to *data_encoding*.

errors may be given to define the error handling. It defaults to `'strict'`, which causes *ValueError* to be raised in case an encoding error occurs.

codecs.**iterencode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
Uses an incremental encoder to iteratively encode the input provided by *iterator*. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental encoder.

This function requires that the codec accept text *str* objects to encode. Therefore it does not support bytes-to-bytes encoders such as `base64_codec`.

codecs.**iterdecode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
Uses an incremental decoder to iteratively decode the input provided by *iterator*. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental decoder.

This function requires that the codec accept *bytes* objects to decode. Therefore it does not support text-to-text encoders such as `rot_13`, although `rot_13` may be used equivalently with *iterencode()*.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

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**
These constants define various byte sequences, being Unicode byte order marks (BOMs) for several encodings. They are used in UTF-16 and UTF-32 data streams to indicate the byte order used, and in UTF-8 as a Unicode signature. *BOM_UTF16* is either *BOM_UTF16_BE* or *BOM_UTF16_LE* depending on the platform's native byte order, *BOM* is an alias for *BOM_UTF16*, *BOM_LE* for *BOM_UTF16_LE* and *BOM_BE* for *BOM_UTF16_BE*. The others represent the BOM in UTF-8 and UTF-32 encodings.

7.2.1 Codec Base Classes

The *codecs* module defines a set of base classes which define the interfaces for working with codec objects, and can also be used as the basis for custom codec implementations.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols. Codec authors also need to define how the codec will handle encoding and decoding errors.

Error Handlers

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument. The following string values are defined and implemented by all standard Python codecs:

Value	Meaning
`'strict'`	Raise *UnicodeError* (or a subclass); this is the default. Implemented in *strict_errors()*.
`'ignore'`	Ignore the malformed data and continue without further notice. Implemented in *ignore_errors()*.

The following error handlers are only applicable to *text encodings*:

Value	Meaning
`'replace'`	Replace with a suitable replacement marker; Python will use the official `U+FFFD` REPLACEMENT CHARACTER for the built-in codecs on decoding, and '?' on encoding. Implemented in *replace_errors()*.
`'xmlcharrefreplace'`	Replace with the appropriate XML character reference (only for encoding). Implemented in *xmlcharrefreplace_errors()*.
`'backslashreplace'`	Replace with backslashed escape sequences. Implemented in *backslashreplace_errors()*.
`'namereplace'`	Replace with `\N{...}` escape sequences (only for encoding). Implemented in *namereplace_errors()*.
`'surrogateescape'`	On decoding, replace byte with individual surrogate code ranging from `U+DC80` to `U+DCFF`. This code will then be turned back into the same byte when the `'surrogateescape'` error handler is used when encoding the data. (See **PEP 383** for more.)

In addition, the following error handler is specific to the given codecs:

Value	Codecs	Meaning
`'surrogatepass'`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding of surrogate codes. These codecs normally treat the presence of surrogates as an error.

버전 3.1에 추가: The `'surrogateescape'` and `'surrogatepass'` error handlers.

버전 3.4에서 변경: The `'surrogatepass'` error handlers now works with utf-16* and utf-32* codecs.

버전 3.5에 추가: The `'namereplace'` error handler.

버전 3.5에서 변경: The `'backslashreplace'` error handlers now works with decoding and translating.

The set of allowed values can be extended by registering a new named error handler:

`codecs.`**`register_error`**(*name*, *error_handler*)

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a *UnicodeEncodeError* instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either *str* or *bytes*. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an *IndexError* will be raised.

Decoding and translating works similarly, except *UnicodeDecodeError* or *UnicodeTranslateError* will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name:

codecs.**lookup_error**(*name*)
Return the error handler previously registered under the name *name*.

Raises a *LookupError* in case the handler cannot be found.

The following standard error handlers are also made available as module level functions:

codecs.**strict_errors**(*exception*)
Implements the `'strict'` error handling: each encoding or decoding error raises a *UnicodeError*.

codecs.**replace_errors**(*exception*)
Implements the `'replace'` error handling (for *text encodings* only): substitutes `'?'` for encoding errors (to be encoded by the codec), and `'\ufffd'` (the Unicode replacement character) for decoding errors.

codecs.**ignore_errors**(*exception*)
Implements the `'ignore'` error handling: malformed data is ignored and encoding or decoding is continued without further notice.

codecs.**xmlcharrefreplace_errors**(*exception*)
Implements the `'xmlcharrefreplace'` error handling (for encoding with *text encodings* only): the unencodable character is replaced by an appropriate XML character reference.

codecs.**backslashreplace_errors**(*exception*)
Implements the `'backslashreplace'` error handling (for *text encodings* only): malformed data is replaced by a backslashed escape sequence.

codecs.**namereplace_errors**(*exception*)
Implements the `'namereplace'` error handling (for encoding with *text encodings* only): the unencodable character is replaced by a `\N{...}` escape sequence.

버전 3.5에 추가.

Stateless Encoding and Decoding

The base `Codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder:

`Codec`.**encode**(*input*[, *errors*])
Encodes the object *input* and returns a tuple (output object, length consumed). For instance, *text encoding* converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use *StreamWriter* for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Codec.**decode**(*input*[, *errors*])

Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a *text encoding*, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface – for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use *StreamReader* for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Incremental Encoding and Decoding

The *IncrementalEncoder* and *IncrementalDecoder* classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the *encode()*/*decode()* method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the *encode()*/*decode()* method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

The *IncrementalEncoder* class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalEncoder**(*errors='strict'*)

Constructor for an *IncrementalEncoder* instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *IncrementalEncoder* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *IncrementalEncoder* object.

encode(*object*[, *final*])

Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to *encode()* *final* must be true (the default is false).

reset()

Reset the encoder to the initial state. The output is discarded: call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()

Return the current state of the encoder which must be an integer. The implementation should make sure that `0` is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the encoder to *state*. *state* must be an encoder state returned by *getstate()*.

IncrementalDecoder Objects

The *IncrementalDecoder* class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalDecoder**(*errors='strict'*)
Constructor for an *IncrementalDecoder* instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *IncrementalDecoder* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *IncrementalDecoder* object.

decode(*object*[, *final*])
Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to *decode()* *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()
Reset the decoder to the initial state.

getstate()
Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that 0 is the most common additional state info.) If this additional state info is 0 it must be possible to set the decoder to the state which has no input buffered and 0 as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate(*state*)
Set the state of the decoder to *state*. *state* must be a decoder state returned by *getstate()*.

Stream Encoding and Decoding

The *StreamWriter* and *StreamReader* classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See encodings.utf_8 for an example of how this is done.

StreamWriter Objects

The *StreamWriter* class is a subclass of Codec and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class codecs.**StreamWriter**(*stream*, *errors='strict'*)
Constructor for a *StreamWriter* instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The `StreamWriter` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamWriter` object.

write(*object*)
Writes the object's contents encoded to the stream.

writelines(*list*)
Writes the concatenated list of strings to the stream (possibly by reusing the `write()` method). The standard bytes-to-bytes codecs do not support this method.

reset()
Flushes and resets the codec buffers used for keeping state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.

StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class codecs.**StreamReader**(*stream*, *errors='strict'*)
Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

read([*size*[, *chars*[, *firstline*]]])
Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

readline([*size*[, *keepends*]])
Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

readlines([*sizehint*[, *keepends*]])
Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's `decode()` method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream's `read()` method.

reset()
Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter Objects

The `StreamReaderWriter` is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)
Creates a `StreamReaderWriter` instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The `StreamRecoder` translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)
Creates a `StreamRecoder` instance which implements a two-way conversion: *encode* and *decode* work on the frontend — the data visible to code calling `read()` and `write()`, while *Reader* and *Writer* work on the backend — the data in *stream*.

You can use these objects to do transparent transcodings, e.g., from Latin-1 to UTF-8 and back.

The *stream* argument must be a file-like object.

The *encode* and *decode* arguments must adhere to the `Codec` interface. *Reader* and *Writer* must be factory functions or classes providing objects of the `StreamReader` and `StreamWriter` interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

StreamRecoder instances define the combined interfaces of *StreamReader* and *StreamWriter* classes. They inherit all other methods and attributes from the underlying stream.

7.2.2 Encodings and Unicode

Strings are stored internally as sequences of code points in range 0x0–0x10FFFF. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising 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 serialisation codecs, which are collectivity referred to as *text encodings*.

The simplest text encoding (called 'latin-1' or 'iso-8859-1') maps the code points 0–255 to the bytes 0x0–0xff, which means that a string object that contains code points above U+00FF can't be encoded with this codec. Doing so will raise a *UnicodeEncodeError* that looks like the following (although the details of the error message may differ): UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes 0x0–0xff. To see how this is done simply open e.g. encodings/cp1252.py (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called UTF-32-BE and UTF-32-LE respectively. Their disadvantage is that if e.g. you use UTF-32-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-32 avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 or UTF-32 byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character U+FEFF. This character can be prepended to every UTF-16 or UTF-32 byte sequence. The byte swapped version of this character (0xFFFE) is an illegal character that may not appear in a Unicode text. So when the first character in an UTF-16 or UTF-32 byte sequence appears to be a U+FFFE the bytes have to be swapped on decoding. Unfortunately the character U+FEFF had a second purpose as a ZERO WIDTH NO-BREAK SPACE: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using U+FEFF as a ZERO WIDTH NO-BREAK SPACE has been deprecated (with U+2060 (WORD JOINER) assuming this role). Nevertheless Unicode software still must be able to handle U+FEFF in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a ZERO WIDTH NO-BREAK SPACE it's a normal character that will be decoded like any other.

There's another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four 1 bits followed by a 0 bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

Range	Encoding
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

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any `U+FEFF` character in the decoded string (even if it's the first character) is treated as a `ZERO WIDTH NO-BREAK SPACE`.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls `"utf-8-sig"`) for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

> LATIN SMALL LETTER I WITH DIAERESIS
> RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
> INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a `utf-8-sig` encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write `0xef`, `0xbb`, `0xbf` as the first three bytes to the file. On decoding `utf-8-sig` will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3 Standard Encodings

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; therefore, e.g. `'utf-8'` is a valid alias for the `'utf_8'` codec.

CPython implementation detail: Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

버전 3.6에서 변경: Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
ascii	646, us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
big5hkscs	big5-hkscs, hkscs	Traditional Chinese
cp037	IBM037, IBM039	English
cp273	273, IBM273, csIBM273	German 버전 3.4에 추가.

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

Codec	Aliases	Languages
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		Arabic
cp737		Greek
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp856		Hebrew
cp857	857, IBM857	Turkish
cp858	858, IBM858	Western Europe
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	Hebrew
cp863	863, IBM863	Canadian
cp864	IBM864	Arabic
cp865	865, IBM865	Danish, Norwegian
cp866	866, IBM866	Russian
cp869	869, CP-GR, IBM869	Greek
cp874		Thai
cp875		Greek
cp932	932, ms932, mskanji, ms-kanji	Japanese
cp949	949, ms949, uhc	Korean
cp950	950, ms950	Traditional Chinese
cp1006		Urdu
cp1026	ibm1026	Turkish
cp1125	1125, ibm1125, cp866u, ruscii	Ukrainian 버전 3.4에 추가.
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	Greek
cp1254	windows-1254	Turkish
cp1255	windows-1255	Hebrew
cp1256	windows-1256	Arabic
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	Vietnamese
cp65001		Windows only: Windows UTF-8 (`CP_UTF8`) 버전 3.3에 추가.
euc_jp	eucjp, ujis, u-jis	Japanese
euc_jis_2004	jisx0213, eucjis2004	Japanese
euc_jisx0213	eucjisx0213	Japanese

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

Codec	Aliases	Languages
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	Simplified Chinese
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	Simplified Chinese
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japanese
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japanese
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japanese
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japanese
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japanese
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Korean
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	Western Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Baltic languages
iso8859_5	iso-8859-5, cyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	Arabic
iso8859_7	iso-8859-7, greek, greek8	Greek
iso8859_8	iso-8859-8, hebrew	Hebrew
iso8859_9	iso-8859-9, latin5, L5	Turkish
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_11	iso-8859-11, thai	Thai languages
iso8859_13	iso-8859-13, latin7, L7	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15, latin9, L9	Western Europe
iso8859_16	iso-8859-16, latin10, L10	South-Eastern Europe
johab	cp1361, ms1361	Korean
koi8_r		Russian
koi8_t		Tajik 버전 3.5에 추가.
koi8_u		Ukrainian
kz1048	kz_1048, strk1048_2002, rk1048	Kazakh 버전 3.5에 추가.
mac_cyrillic	maccyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
mac_greek	macgreek	Greek
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope	Central and Eastern Europe
mac_roman	macroman, macintosh	Western Europe
mac_turkish	macturkish	Turkish

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

Codec	Aliases	Languages
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japanese
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japanese
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japanese
utf_32	U32, utf32	all languages
utf_32_be	UTF-32BE	all languages
utf_32_le	UTF-32LE	all languages
utf_16	U16, utf16	all languages
utf_16_be	UTF-16BE	all languages
utf_16_le	UTF-16LE	all languages
utf_7	U7, unicode-1-1-utf-7	all languages
utf_8	U8, UTF, utf8	all languages
utf_8_sig		all languages

버전 3.4에서 변경: The utf-16* and utf-32* encoders no longer allow surrogate code points (`U+D800`–`U+DFFF`) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

7.2.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated meaning describes the encoding direction.

Text Encodings

The following codecs provide `str` to `bytes` encoding and *bytes-like object* to `str` decoding, similar to the Unicode text encodings.

Codec	Aliases	Meaning
idna		Implement **RFC 3490**, see also *encodings.idna*. Only `errors='strict'` is supported.
mbcs	ansi, dbcs	Windows only: Encode the operand according to the ANSI codepage (CP_ACP).
oem		Windows only: Encode the operand according to the OEM codepage (CP_OEMCP). 버전 3.6에 추가.
palmos		Encoding of PalmOS 3.5.
punycode		Implement **RFC 3492**. Stateful codecs are not supported.
raw_unicode_escape		Latin-1 encoding with `\uXXXX` and `\UXXXXXXXX` for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decode from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.
unicode_internal		Return the internal representation of the operand. Stateful codecs are not supported. 버전 3.3부터 폐지: This representation is obsoleted by **PEP 393**.

Binary Transforms

The following codecs provide binary transforms: *bytes-like object* to *bytes* mappings. They are not supported by *bytes.decode()* (which only produces *str* output).

Codec	Aliases	Meaning	Encoder / decoder
base64_codec[1]	base64, base_64	Convert the operand to multiline MIME base64 (the result always includes a trailing `'\n'`). 버전 3.4에서 변경: accepts any *bytes-like object* as input for encoding and decoding	*base64.encodebytes()* / *base64.decodebytes()*
bz2_codec	bz2	Compress the operand using bz2.	*bz2.compress()* / *bz2.decompress()*
hex_codec	hex	Convert the operand to hexadecimal representation, with two digits per byte.	*binascii.b2a_hex()* / *binascii.a2b_hex()*
quopri_codec	quopri, quotedprint-able, quoted_printable	Convert the operand to MIME quoted printable.	*quopri.encode()* with `quotetabs=True` / *quopri.decode()*
uu_codec	uu	Convert the operand using uuencode.	*uu.encode()* / *uu.decode()*
zlib_codec	zip, zlib	Compress the operand using gzip.	*zlib.compress()* / *zlib.decompress()*

버전 3.2에 추가: Restoration of the binary transforms.

버전 3.4에서 변경: Restoration of the aliases for the binary transforms.

Text Transforms

The following codec provides a text transform: a *str* to *str* mapping. It is not supported by *str.encode()* (which only produces *bytes* output).

Codec	Aliases	Meaning
rot_13	rot13	Return the Caesar-cypher encryption of the operand.

버전 3.2에 추가: Restoration of the `rot_13` text transform.

버전 3.4에서 변경: Restoration of the `rot13` alias.

7.2.5 `encodings.idna` — Internationalized Domain Names in Applications

This module implements **RFC 3490** (Internationalized Domain Names in Applications) and **RFC 3492** (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the `punycode` encoding and *stringprep*.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP *Host* fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

[1] In addition to *bytes-like objects*, `'base64_codec'` also accepts ASCII-only instances of *str* for decoding

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the *socket* module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as *http.client* and *ftplib*, accept Unicode host names (*http.client* then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: applications wishing to present such host names to the user should decode them to Unicode.

The module *encodings.idna* also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

`encodings.idna.`**`nameprep`**(*label*)
Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is true.

`encodings.idna.`**`ToASCII`**(*label*)
Convert a label to ASCII, as specified in **RFC 3490**. `UseSTD3ASCIIRules` is assumed to be false.

`encodings.idna.`**`ToUnicode`**(*label*)
Convert a label to Unicode, as specified in **RFC 3490**.

7.2.6 `encodings.mbcs` — Windows ANSI codepage

This module implements the ANSI codepage (CP_ACP).

Availability: Windows only.

버전 3.3에서 변경: Support any error handler.

버전 3.2에서 변경: Before 3.2, the *errors* argument was ignored; `'replace'` was always used to encode, and `'ignore'` to decode.

7.2.7 `encodings.utf_8_sig` — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec. On encoding, a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). On decoding, an optional UTF-8 encoded BOM at the start of the data will be skipped.

CHAPTER 8

데이터형

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, double-ended queues, and enumerations.

Python also provides some built-in data types, in particular, *dict*, *list*, *set* and *frozenset*, and *tuple*. The *str* class is used to hold Unicode strings, and the *bytes* and *bytearray* classes are used to hold binary data.

이 장에서는 다음 모듈에 관해 설명합니다:

8.1 datetime — 기본 날짜와 시간 형

소스 코드: Lib/datetime.py

datetime 모듈은 날짜와 시간을 간단하거나 복잡한 방법으로 조작하는 클래스를 제공합니다. 날짜와 시간 산술이 지원되지만, 구현의 초점은 출력 포매팅과 조작을 위한 효율적인 어트리뷰트 추출입니다. 관련 기능에 대해서는, *time*과 *calendar* 모듈도 참조하십시오.

날짜와 시간 객체에는 두 가지 종류가 있습니다: "나이브(naive)"와 "어웨어(aware)".

어웨어 객체는 다른 어웨어 객체와의 상대적인 위치를 파악하기 위한, 시간대와 일광 절약 시간 정보와 같은 적용 가능한 알고리즘과 정치적 시간 조정에 대한 충분한 지식을 갖추고 있습니다. 어웨어 객체는 자의적으로 해석할 여지 없는 특정 시간을 나타내기 위해 사용됩니다[1].

나이브 객체는 모호하지 않게 자신과 다른 날짜/시간 객체의 상대적인 위치를 파악할 수 있는 충분한 정보를 포함하지 않습니다. 나이브 객체가 UTC(Coordinated Universal Time), 지역 시간 또는 다른 시간대의 시간 중 어느 것을 나타내는지는 순전히 프로그램에 달려있습니다. 특정 숫자가 미터, 마일 또는 질량 중 어는 것을 나타내는지가 프로그램에 달린 것과 마찬가지입니다. 나이브 객체는 이해하기 쉽고 작업하기 쉽지만, 현실의 일부 측면을 무시하는 대가를 치릅니다.

어웨어 객체가 필요한 응용 프로그램을 위해, *datetime* 과 *time* 객체에는 추상 *tzinfo* 클래스의 서브 클래스 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 보관합니

[1] 즉, 상대론적 효과를 무시한다면

다. *datetime* 모듈에서는 오직 하나의 구상 *tzinfo* 클래스, *timezone* 클래스만 제공됨에 유의하십시오. *timezone* 클래스는 UTC 자체나 북미 EST와 EDT 시간대와 같은 UTC로부터 고정 오프셋을 갖는 간단한 시간대를 나타낼 수 있습니다. 더욱 세부적인 수준의 시간대 지원은 응용 프로그램에 달려 있습니다. 전 세계의 시간 조정에 대한 규칙은 합리적이라기보다 정치적이고, 자주 변경되며, UTC 이외에 모든 응용 프로그램에 적합한 표준은 없습니다.

datetime 모듈은 다음 상수를 내보냅니다:

datetime.**MINYEAR**
: *date*나 *datetime* 객체에서 허용되는 가장 작은 연도 번호. *MINYEAR*는 1입니다.

datetime.**MAXYEAR**
: *date*나 *datetime* 객체에서 허용되는 가장 큰 연도 번호. *MAXYEAR*는 9999입니다.

더 보기:

모듈 *calendar* 일반 달력 관련 함수들.

모듈 *time* 시간 액세스와 변환.

8.1.1 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**
: 두 *date*, *time* 또는 *datetime* 인스턴스 간의 차이를 마이크로초 해상도로 나타내는 기간.

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.

  버전 3.2에 추가.

이러한 형의 객체는 불변입니다.

date 형의 객체는 항상 나이브합니다.

*time*나 *datetime* 형의 객체는 나이브하거나 어웨어할 수 있습니다. *datetime* 객체 *d*는 `d.tzinfo`가 `None`이 아니고, `d.tzinfo.utcoffset(d)`가 `None`을 반환하지 않으면 어웨어합니다. `d.tzinfo`가 `None`이거나, `d.tzinfo`는 `None`이 아니지만 `d.tzinfo.utcoffset(d)`가 `None`을 반환하면 *d*는 나이브합니다. *time* 객체 *t*는 `t.tzinfo``가 ``None`이 아니고 `t.tzinfo.utcoffset(None)`이 `None`을 반환하지 않으면 어웨어합니다. 그렇지 않으면, *t*는 나이브합니다.

나이브와 어웨어 간의 차이점은 *timedelta* 객체에는 적용되지 않습니다.

서브 클래스 관계:

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

8.1.2 timedelta 객체

timedelta 객체는 두 날짜나 시간의 차이인 기간을 나타냅니다.

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

모든 인자는 선택적이며 기본값은 0입니다. 인자는 정수나 부동 소수점 수일 수 있으며, 양수나 음수일 수 있습니다.

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`

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)
```

클래스 어트리뷰트는 다음과 같습니다:

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)`.

정규화로 인해, `timedelta.max` > `-timedelta.min`입니다. `-timedelta.max`는 *timedelta* 객체로 표현할 수 없습니다.

인스턴스 어트리뷰트 (읽기 전용):

어트리뷰트	값
`days`	-999999999와 999999999 사이, 경계 포함
`seconds`	0과 86399 사이, 경계 포함
`microseconds`	0과 999999 사이, 경계 포함

지원되는 연산:

연산	결과
`t1 = t2 + t3`	*t2*와 *t3*의 합. 이후에는 *t1*-*t2* == *t3* 과 *t1*-*t3* == *t2*가 참입니다. (1)
`t1 = t2 - t3`	*t2*와 *t3*의 차이. 이후에는 *t1* == *t2* - *t3* 과 *t2* == *t1* + *t3*가 참입니다. (1)(6)
`t1 = t2 * i` 또는 `t1 = i * t2`	델타에 정수를 곱합니다. 이후에는 `i != 0`일 때, *t1* // i == *t2*가 참입니다.
	일반적으로, *t1* * i == *t1* * (i-1) + *t1*은 참입니다. (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`	전체 기간 *t2*를 구간 단위 *t3*으로 나누기 (3). *float* 객체를 반환합니다.
`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)`	몫과 나머지를 계산합니다: `q = t1 // t2` (3) 과 `r = t1 % t2`. q는 정수고 r은 *timedelta* 객체입니다.
`+t1`	같은 값을 갖는 *timedelta* 객체를 반환합니다. (2)
`-t1`	*timedelta*(*-t1.days*, *-t1.seconds*, *-t1.microseconds*) 와 *t1** -1과 동등합니다. (1)(4)
`abs(t)`	`t.days >= 0` 일 때 *+t* 와 `t.days < 0` 일 때 *-t*와 동등합니다. (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) 0으로 나누면 *ZeroDivisionError*가 발생합니다.

(4) *-timedelta.max*는 *timedelta* 객체로 표현할 수 없습니다.

(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 객체의 비교가 지원되는데, 더 짧은 기간을 나타내는 *timedelta* 객체를 더 작은 것으로 간주합니다. 혼합형 비교가 객체 주소 기반의 기본 비교로 떨어지는 것을 막기 위해, *timedelta* 객체가 다른 형의 객체와 비교될 때, 비교가 == 이나 !=가 아니면 *TypeError*가 발생합니다. 두 상황에 해당하면 각각 *False* 나 *True*를 반환합니다.

timedelta 객체는 해시 가능(딕셔너리 키로 사용 가능)하고, 효율적인 피클링을 지원하며, 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.

버전 3.2에 추가.

사용 예:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(days=3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(days=3285), 9)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3 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 부릅니다. 이것은 Dershowitz와 Reingold의 책 Calendrical Calculations에 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.

class datetime.**date**(*year*, *month*, *day*)
All arguments are required. Arguments must be integers in the following ranges:

- `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:

classmethod date.**today**()
현재 지역 날짜를 반환합니다. 이것은 `date.fromtimestamp(time.time())`와 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. `1 <= ordinal <= date.max.toordinal()`이 아니면 *ValueError*가 발생합니다. 모든 date *d*에 대해, `date.fromordinal(d.toordinal()) == d`입니다.

classmethod date.**fromisoformat**(*date_string*)
*date.isoformat()*으로 만든 형식의 *date_string*에 해당하는 *date*를 반환합니다. 구체적으로, 이 함수는 `YYYY-MM-DD` 형식의 문자열을 지원합니다.

> **조심:** 이것은 임의의 ISO 8601 문자열을 구문 분석하는 것을 지원하지 않습니다 - 이것은 *date.isoformat()*의 역연산이고자 할 뿐입니다.

버전 3.7에 추가.

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*는 *date1*에서 `timedelta.days` 일 이동한 날짜입니다. (1)
`date2 = date1 - timedelta`	`date2 + timedelta == date1`가 성립하는 *date2*를 계산합니다. (2)
`timedelta = date1 - date2`	(3)
`date1 < date2`	*date1*이 *date2*에 앞서면 *date1*는 *date2*보다 작은 것으로 간주합니다. (4)

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.microseconds`는 무시됩니다. `date2.year`가 *MINYEAR*보다 작거나 *MAXYEAR*보다 크게 되려고 하면 *OverflowError*가 발생합니다.

(2) `timedelta.seconds`와 `timedelta.microseconds`는 무시됩니다.

(3) 이것은 정확하고, 오버플로 할 수 없습니다. timedelta.seconds와 timedelta.microseconds는 0이고, 이후에 date2 + timedelta == date1 이 됩니다.

(4) 즉, 오직 `date1.toordinal() < date2.toordinal()`일 때만 `date1 < date2`입니다. 비교 대상이 *date* 객체가 아니면 날짜 비교는 *TypeError*를 발생시킵니다. 그러나, 비교 대상에 `timetuple()` 어트리뷰트가 있으면, 대신 `NotImplemented`가 반환됩니다. 이 훅은 다른 형의 날짜 객체가 혼합형 비교를 구현할 기회를 제공합니다. 그렇지 않으면, *date* 객체가 다른 형의 객체와 비교될 때, 비교가 `==`나 `!=`가 아니면 *TypeError*가 발생합니다. 두 상황에 해당하면 각각 *False* 나 *True*를 반환합니다

날짜는 딕셔너리 키로 사용할 수 있습니다. 불리언 문맥에서, 모든 *date* 객체는 참으로 간주합니다.

인스턴스 메서드:

date.**replace**(*year=self.year*, *month=self.month*, *day=self.day*)
키워드 인자로 새로운 값이 주어진 매개 변수들을 제외하고, 같은 값을 가진 date를 반환합니다. 예를 들어, `d == date(2002, 12, 31)`이면, `d.replace(day=26) == date(2002, 12, 26)`입니다.

date.**timetuple**()
*time.localtime()*이 반환하는 것과 같은 *time.struct_time*을 반환합니다. 시, 분 및 초는 0이고, DST 플래그는 -1입니다. `d.timetuple()`은 `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))`와 동등합니다. 여기서 `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1`은 1월 1일에 1로 시작하는 현재 연도의 날짜 번호입니다.

date.**toordinal**()
역산 그레고리력 서수를 돌려줍니다. 1년 1월 1일의 서수는 1입니다. 임의의 *date* 객체 *d*에 대해 `date.fromordinal(d.toordinal()) == d`입니다.

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**()
3-튜플 (ISO 연도, ISO 주 번호, ISO 요일)을 반환합니다.

ISO 달력은 그레고리력의 널리 사용되는 변형입니다. https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm 에 잘 설명되어 있습니다.

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. 그래서 `date(2003, 12, 29).isocalendar() == (2004, 1, 1)`이고 `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`입니다.

date.**isoformat**()
ISO 8601 형식으로 날짜를 나타내는 문자열을 반환합니다, 'YYYY-MM-DD'. 예를 들어, `date(2002, 12, 4).isoformat() == '2002-12-04'`.

date.**__str__**()
날짜 *d*에 대해, `str(d)`는 `d.isoformat()`와 동등합니다.

date.**ctime**()
날짜를 나타내는 문자열을 반환합니다, 예를 들어 `date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'`. `d.ctime()`은 네이티브 C `ctime()` 함수(*time.ctime()*은 호출하지만 *date.ctime()*은 호출하지 않습니다)가 C 표준을 준수하는 플랫폼에서 `time.ctime(time.mktime(d.timetuple()))`와 동등합니다.

date.**strftime**(*format*)
명시적인 포맷 문자열로 제어되는, 날짜를 나타내는 문자열을 반환합니다. 시, 분 또는 초를 나타내는 포맷 코드는 0 값을 보게 됩니다. 포매팅 지시자의 전체 목록은, *strftime()과 strptime() 동작*를 참조하십시오.

date.**__format__**(*format*)
*date.strftime()*과 같습니다. 이것이 포맷 문자열 리터럴과 *str.format()*을 사용할 때 *date* 객체를 위한 포맷 문자열을 지정할 수 있도록 합니다. 포매팅 지시자의 전체 목록은 *strftime()과 strptime() 동작*를 참조하십시오.

이벤트까지 남은 날 수 계산 예:

```
>>> 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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> time_to_birthday.days
202
```

*date*로 작업하는 예:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> 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 )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4 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*)

The year, month and day arguments are required. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments must be integers in the following ranges:

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <=` 주어진 `month`와 `year`에서의 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 3.6에 추가: `fold` 인자가 추가되었습니다.

다른 생성자, 모든 클래스 메서드:

classmethod datetime.**today**()
*tzinfo*가 None인 현재 지역 datetime을 반환합니다. 이것은 `datetime.fromtimestamp(time.time())`과 동등합니다. *now()*, *fromtimestamp()*를 참조하십시오.

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. 이때 결과는 `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`와 동등합니다. *today()*, *utcnow()*도 참조하십시오.

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.

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. 이때 결과는 `tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))`와 동등합니다.

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. *utcfromtimestamp()*도 참조하십시오.

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. 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.

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.

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 None입니다.

classmethod datetime.**combine**(*date*, *time*, *tzinfo=self.tzinfo*)

지정된 *date* 객체와 같은 날짜 구성 요소와 지정된 *time* 객체와 같은 시간 구성 요소를 갖는 새 *datetime* 객체를 반환합니다. *tzinfo* 인자가 제공되면, 그 값은 결과의 *tzinfo* 어트리뷰트를 설정하는 데 사용되며, 그렇지 않으면 *time* 인자의 *tzinfo* 어트리뷰트가 사용됩니다.

모든 *datetime* 객체 *d*에 대해, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`가 성립합니다. date가 *datetime* 객체면, 그것의 시간 구성 요소와 *tzinfo* 어트리뷰트가 무시됩니다.

버전 3.6에서 변경: *tzinfo* 인자가 추가되었습니다.

classmethod datetime.**fromisoformat**(*date_string*)

*date.isoformat()*과 *datetime.isoformat()*이 출력하는 형식 중 하나인 *date_string*에 해당하는 *datetime*을 반환합니다. 구체적으로, 이 함수는 `YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]]` 형식의 문자열을 지원합니다. 여기서 `*`는 임의의 단일 문자와 일치 할 수 있습니다.

> **조심:** This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of *datetime.isoformat()*. A more full-featured ISO 8601 parser, `dateutil.parser.isoparse` is available in the third-party package dateutil.

버전 3.7에 추가.

classmethod datetime.**strptime**(*date_string*, *format*)

*format*에 따라 구문 분석된, *date_string*에 해당하는 *datetime*를 반환합니다. 이것은 `datetime(*(time.strptime(date_string, format)[0:6]))`과 동등합니다. date_string과 format을 *time.strptime()*로 구문 분석할 수 없거나, 시간 튜플이 아닌 값을 반환하면 *ValueError*가 발생합니다. 포매팅 지시자의 전체 목록은 *strftime()과 strptime() 동작*을 참조하십시오.

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 None입니다.

datetime.**fold**
`[0, 1]` 범위입니다. 반복되는 구간 동안 벽 시간(wall time)의 모호함을 제거하는 데 사용됩니다. 반복되는 구간은 일광 절약 시간이 끝날 때나 현재 지역의 UTC 오프셋이 정치적인 이유로 줄어들어 시계를 되돌릴 때 발생합니다. 값 0(1)은 같은 벽 시간을 나타내는 두 순간 중 이전(이후)을 나타냅니다.

버전 3.6에 추가.

지원되는 연산:

연산	결과
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 < datetime2`	*datetime*과 *datetime*을 비교합니다. (4)

(1) datetime2는 datetime1에서 timedelta 기간만큼 이동한 시간이며, `timedelta.days` > 0이면 미래로, `timedelta.days` < 0이면 과거로 이동합니다. 결과는 입력 datetime과 같은 *tzinfo* 어트리뷰트를 가지고, 이후에 datetime2 - datetime1 == timedelta 입니다. datetime2.year가 *MINYEAR*보다 작거나 *MAXYEAR*보다 커지려고 하면 *OverflowError*가 발생합니다. 입력이 어웨어 객체일 때도 시간대 조정이 수행되지 않음에 유의하십시오.

(2) datetime2 + timedelta == datetime1 을 만족하는 datetime2를 계산합니다. 덧셈과 마찬가지로, 결과는 입력 datetime과 같은 *tzinfo* 어트리뷰트를 가지며 입력이 어웨어일 때도 시간대 조정이 수행되지 않습니다.

(3) *datetime*에서 *datetime*을 빼는 것은 두 피연산자 모두 나이브하거나, 모두 어웨어할 때만 정의됩니다. 하나가 어웨어이고 다른 하나가 나이브면, *TypeError*가 발생합니다.

둘 다 나이브하거나 둘 다 어웨어하고 같은 *tzinfo* 어트리뷰트를 가지면, *tzinfo* 어트리뷰트는 무시되고 결과는 `datetime2 + t == datetime1` 이 되도록 하는 *timedelta* 객체 *t*입니다. 이때 시간대 조정이 수행되지 않습니다.

둘 다 어웨어하고 *tzinfo* 어트리뷰트가 다르면, `a-b`는 *a*와 *b*가 먼저 나이브 UTC datetime으로 먼저 변환된 것처럼 작동합니다. 구현이 절대 오버플로 하지 않는다는 것을 제외하면 결과는 `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` 입니다.

(4) *datetime1*이 *datetime2*에 앞서면 *datetime1*은 *datetime2*보다 작은 것으로 간주합니다.

하나의 비교 피연산자가 나이브하고 다른 하나는 어웨어하면, 순서 비교가 시도될 때 *TypeError*가 발생합니다. 동등(equality) 비교에서는, 나이브 인스턴스는 절대 어웨어 인스턴스와 같지 않습니다.

비교 피연산자가 모두 어웨어하고, 같은 *tzinfo* 어트리뷰트를 가지면, 공통 *tzinfo* 어트리뷰트가 무시되고 기본 datetime이 비교됩니다. 두 비교 피연산자가 모두 어웨어하고 다른 *tzinfo* 어트리뷰트를 가지면, 비교 피연산자들은 먼저 그들의 UTC 오프셋(`self.utcoffset()` 에서 얻습니다)을 뺀 값으로 조정됩니다.

버전 3.3에서 변경: 나이브와 어웨어 *datetime* 인스턴스 간의 동등 비교는 *TypeError*를 발생시키지 않습니다.

참고: 비교가 객체 주소 기반의 기본 비교 체계로 떨어지는 것을 막기 위해, datetime 비교는 다른 비교 피연산자가 *datetime* 객체가 아니면 일반적으로 *TypeError*를 발생시킵니다. 그러나, 다른 비교 피연산자에 `timetuple()` 어트리뷰트가 있으면 `NotImplemented`가 대신 반환됩니다. 이 훅은 다른 형의 날짜 객체에 혼합형 비교를 구현할 기회를 제공합니다. 그렇지 않으면, *datetime* 객체가 다른 형의 객체와 비교될 때, 비교가 == 나 !=가 아니면 *TypeError*가 발생합니다. 두 상황에 해당하면 각각 *False* 나 *True*를 반환합니다.

---

datetime 객체는 딕셔너리 키로 사용할 수 있습니다. 불리언 문맥에서, 모든 *datetime* 객체는 참으로 간주합니다.

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*`)`
: 키워드 인자로 새로운 값이 주어진 어트리뷰트를 제외하고, 같은 어트리뷰트를 가진 datetime을 반환합니다. `tzinfo=None`을 지정하면 날짜와 시간 데이터의 변환 없이 어웨어 datetime에서 나이브 datetime을 만들 수 있습니다.

  버전 3.6에 추가: `fold` 인자가 추가되었습니다.

`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.

  제공된다면 *tz*는 *tzinfo* 서브 클래스의 인스턴스여야 하며, *utcoffset()*과 *dst()* 메서드는 `None`을 반환하지 않아야 합니다. *self*가 나이브하면, 시스템 시간대의 시간을 나타내는 것으로 가정합니다.

  인자 없이 (또는 `tz=None`으로) 호출되면 대상 시간대는 시스템 시간대로 간주합니다. 변환된 datetime 인스턴스의 `.tzinfo` 어트리뷰트는 OS에서 얻은 시간대 이름과 오프셋을 사용하는 *timezone*의 인스턴스로 설정됩니다.

  `self.tzinfo`가 *tz*면, `self.astimezone(tz)`는 *self*와 같습니다: 날짜나 시간 데이터 조정이 수행되지 않습니다. 그렇지 않으면 결과는 *self*와 같은 UTC 시간을 나타내는 *tz* 시간대의 지역 시간입니다: `astz = dt.astimezone(tz)` 후에, `astz - astz.utcoffset()`는 `dt - dt.utcoffset()`과 같은 날짜와 시간 데이터를 갖습니다.

날짜와 시간 데이터를 조정하지 않고 시간대 객체 *tz*를 datetime *dt*에 연결하기만 하려면, `dt.replace(tzinfo=tz)`를 사용하십시오. 날짜와 시간 데이터를 변환하지 않고 어웨어 datetime *dt*에서 시간대 객체를 제거하려면, `dt.replace(tzinfo=None)`를 사용하십시오.

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 time zone 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 `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst))`와 동등합니다. 여기서 `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1`은 1월 1일에 1로 시작하는 현재 연도의 날짜 번호입니다. 결과의 `tm_isdst` 플래그는 *dst()* 메서드에 따라 설정됩니다: *tzinfo*가 None이거나 *dst()*가 None을 반환하면, `tm_isdst`는 -1로 설정됩니다; 그렇지 않고 *dst()*가 0이 아닌 값을 반환하면, `tm_isdst`는 1로 설정됩니다; 그렇지 않으면 `tm_isdst`는 0으로 설정됩니다.

datetime.**utctimetuple**()
datetime 인스턴스 *d*가 나이브하면, 이것은 `d.dst()`가 무엇을 반환하는지와 관계없이 `tm_isdst`가 강제로 0이 된다는 점만 제외하면, `d.timetuple()`과 같습니다. DST는 UTC 시간에는 적용되지 않습니다.

*d*가 어웨어하면, *d*는 `d.utcoffset()`을 빼서 UTC 시간으로 정규화되고, 정규화된 시간의 *time.struct_time*이 반환됩니다. `tm_isdst`는 강제로 0이 됩니다. *d*.year가 MINYEAR나 MAXYEAR고 UTC 조정이 연도 경계를 넘어가면 *OverflowError*가 발생할 수 있습니다.

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.

나이브 *datetime* 인스턴스는 지역 시간을 나타내는 것으로 간주하며 이 메서드는 변환을 수행하기 위해 플랫폼 C `mktime()` 함수에 의존합니다. *datetime*는 많은 플랫폼에서 `mktime()`보다 더 넓은 범위의 값을 지원하기 때문에, 이 메서드는 먼 과거나 먼 미래의 시간에 대해 *OverflowError*를 발생시킬 수 있습니다.

어웨어 *datetime* 인스턴스의 경우, 반환 값은 다음과 같이 계산됩니다:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

버전 3.3에 추가.

버전 3.6에서 변경: *timestamp()* 메서드는 *fold* 어트리뷰트를 사용하여 반복되는 구간의 시간을 구분합니다.

참고: UTC 시간을 나타내는 나이브 *datetime* 인스턴스에서 직접 POSIX 타임스탬프를 얻는 메서드는 없습니다. 응용 프로그램에서 이 관례를 사용하고 시스템 시간대가 UTC로 설정되어 있지 않으면, `tzinfo=timezone.utc`를 제공하여 POSIX 타임스탬프를 얻을 수 있습니다:

```
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**()
3-튜플 (ISO 연도, ISO 주 번호, ISO 요일)을 반환합니다. `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, 또는 *microsecond*가 0이면, YYYY-MM-DDTHH:MM:SS

*utcoffset()*이 `None`을 반환하지 않으면, UTC 오프셋을 제공하는 문자열을 덧붙입니다: YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]], 또는 *microsecond*가 0이면, YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]].

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):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

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 3.6에 추가: *timespec* 인자가 추가되었습니다.

`datetime.__str__()`
datetime 인스턴스 *d*에 대해, `str(d)`는 `d.isoformat(' ')`과 동등합니다.

`datetime.ctime()`
날짜와 시간을 나타내는 문자열을 반환합니다, 예를 들어 `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'`. `d.ctime()`은 네이티브 C `ctime()` 함수(*time.ctime()*이 호출하지만, *datetime.ctime()*은 호출하지 않습니다)가 C 표준을 준수하는 플랫폼에서 `time.ctime(time.mktime(d.timetuple()))`과 동등합니다.

`datetime.strftime(format)`
명시적인 포맷 문자열에 의해 제어되는 날짜와 시간을 나타내는 문자열을 반환합니다. 포매팅 지시자의 전체 목록은 *strftime()과 strptime() 동작*을 참조하십시오.

`datetime.__format__(format)`
*datetime.strftime()*과 같습니다. 이것이 포맷 문자열 리터럴과 *str.format()*을 사용할 때 *datetime* 객체를 위한 포맷 문자열을 지정할 수 있도록 합니다. 포매팅 지시자의 전체 목록은 *strftime()과 strptime() 동작*을 참조하십시오.

datetime 객체로 작업하는 예제:

```
>>> from datetime import datetime, date, time
>>> # 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() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # 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
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 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.'
```

tzinfo와 함께 datetime 사용하기:

```
>>> 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):
...             # 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):
...         # 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
...         # 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")
...
...         if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
...             return dt + timedelta(hours=4, minutes=30)
...         else:
...             return dt + timedelta(hours=4)
...
...     def dst(self, dt):
...         return timedelta(0)
...
...     def tzname(self, dt):
...         if dt >= self.UTC_MOVE_DATE:
...             return "+04:30"
...         else:
...             return "+04"
...
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> 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.utctimetuple() == dt3.utctimetuple()
True
```

8.1.5 `time` 객체

time 객체는 특정 날짜와 관계없는 (지역) 시간을 나타내며, *tzinfo* 객체를 통해 조정할 수 있습니다.

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

All arguments are optional. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments must be integers in the following ranges:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,

- `fold in [0, 1]`.

이 범위를 벗어나는 인자가 주어지면, *ValueError*가 발생합니다. *tzinfo*의 기본값은 *None*이고, 그 외의 모든 기본값은 `0`입니다.

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`**
`[0, 1]` 범위입니다. 반복되는 구간 동안 벽 시간(wall time)의 모호함을 제거하는 데 사용됩니다. 반복되는 구간은 일광 절약 시간이 끝날 때나 현재 지역의 UTC 오프셋이 정치적인 이유로 줄어들어 시계를 되돌릴 때 발생합니다. 값 0(1)은 같은 벽 시간을 나타내는 두 순간 중 이전(이후)을 나타냅니다.

버전 3.6에 추가.

지원되는 연산:

- *time*과 *time*의 비교, 이때 *a*가 *b*에 앞서면 *a*가 *b*보다 작은 것으로 간주합니다. 하나의 비교 피연산자가 나이브하고 다른 하나는 어웨어하면, 순서 비교가 시도될 때 *TypeError*가 발생합니다. 동등(equality) 비교에서는, 나이브 인스턴스는 절대 어웨어 인스턴스와 같지 않습니다.

  비교 피연산자가 모두 어웨어하고, 같은 *tzinfo* 어트리뷰트를 가지면, 공통 *tzinfo* 어트리뷰트가 무시되고 기본 time이 비교됩니다. 두 비교 피연산자가 모두 어웨어하고 다른 *tzinfo* 어트리뷰트를 가지면, 비교 피연산자들은 먼저 그들의 UTC 오프셋 (`self.utcoffset()`에서 얻습니다)을 뺀 값으로 조정됩니다. 혼합형 비교가 객체 주소 기반의 기본 비교로 떨어지는 것을 막기 위해, *time* 객체가 다른 형의 객체와 비교될 때, 비교가 == 이나 !=가 아니면 *TypeError*가 발생합니다. 두 상황에 해당하면 각각 *False* 나 *True*를 반환합니다.

  버전 3.3에서 변경: 나이브와 어웨어 *time* 인스턴스 간의 동등 비교는 *TypeError*를 발생시키지 않습니다.

- hash, 딕셔너리 키로 사용
- 효율적인 피클링

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.

기타 생성자:

classmethod time.**fromisoformat**(*time_string*)

*time.isoformat()*이 출력하는 형식 중 하나인 *time_string*에 해당하는 *time*을 반환합니다. 구체적으로, 이 함수는 `HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]` 형식의 문자열을 지원합니다.

> **조심:** 이것은 임의의 ISO 8601 문자열을 구문 분석하는 것을 지원하지 않습니다 - 이것은 *time.isoformat()*의 역연산이고자 할 뿐입니다.

버전 3.7에 추가.

인스턴스 메서드:

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ** fold=0*)

키워드 인자로 새로운 값이 주어진 어트리뷰트를 제외하고, 같은 값을 가진 *time*을 반환합니다. `tzinfo=None`을 지정하면 시간 데이터의 변환 없이 어웨어 *time*에서 나이브 *time*을 만들 수 있습니다.

버전 3.6에 추가: `fold` 인자가 추가되었습니다.

time.**isoformat**(*timespec='auto'*)

ISO 8601 형식으로 시간을 나타내는 문자열을 반환합니다, HH:MM:SS.ffffff, 또는 *microsecond*가 0이면, HH:MM:SS *utcoffset()*이 `None`을 반환하지 않으면, UTC 오프셋을 제공하는 문자열을 덧붙입니다: HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]], 또는 self.microsecond가 0이면, HH:MM:SS+HH:MM[:SS[.ffffff]].

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 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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

버전 3.6에 추가: *timespec* 인자가 추가되었습니다.

time.**__str__**()
time *t*에 대해, `str(t)`는 `t.isoformat()`과 동등합니다.

time.**strftime**(*format*)
명시적인 포맷 문자열로 제어되는, 시간을 나타내는 문자열을 반환합니다. 포매팅 지시자의 전체 목록은, *strftime()과 strptime() 동작*을 참조하십시오.

time.**__format__**(*format*)
*time.strftime()*과 같습니다. 이것이 포맷 문자열 리터럴과 *str.format()*을 사용할 때 *time* 객체를 위한 포맷 문자열을 지정할 수 있도록 합니다. 포매팅 지시자의 전체 목록은 *strftime()과 strptime() 동작*을 참조하십시오.

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.

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'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 tzinfo 객체

class datetime.**tzinfo**

이것은 추상 베이스 클래스입니다. 즉, 이 클래스를 직접 인스턴스로 만들면 안 됩니다. 여러분은 구상(concrete) 서브 클래스를 파생시킬 필요가 있고, (적어도) 여러분이 사용하는 *datetime* 메서드에 필요한 표준 *tzinfo* 메서드의 구현을 제공해야 합니다. *datetime* 모듈은 간단한 *tzinfo*의 구상 서브 클래스 *timezone*를 제공하는데, UTC 자체나 북미 EST, EDT와 같은 UTC로부터의 고정 오프셋을 갖는 시간대를 나타낼 수 있습니다.

*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 UTC로부터의 오프셋, 시간대 이름 및 DST 오프셋을 모두 전달된 날짜나 시간 객체에 상대적으로 얻는 메서드들을 지원합니다.

피클링을 위한 특별한 요구 사항: *tzinfo* 서브 클래스는 인자 없이 호출할 수 있는 `__init__()` 메서드를 가져야 합니다. 그렇지 않으면 피클 될 수는 있지만, 다시 역 피클 될 수는 없습니다. 이것은 기술적 요구사항으로, 미래에 완화될 수 있습니다.

*tzinfo*의 구상 서브 클래스는 다음 메서드를 구현해야 할 수도 있습니다. 정확히 어떤 메서드가 필요한지는 어웨어 *datetime* 객체를 사용하는 방법에 따라 다릅니다. 확실하지 않으면, 그냥 모두 구현하십시오.

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. 이 값은 UTC로부터의 총 오프셋입니다: 예를 들어, *tzinfo* 객체가 시간대와 DST 조정을 모두 나타내면, *utcoffset()*은 그들의 합계를 반환해야 합니다. UTC 오프셋을 알 수 없으면, None을 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. DST가 적용되고 있지 않으면, `timedelta(0)`를 반환합니다. DST가 적용 중이면, 오프셋을 *timedelta* 객체로 반환합니다(자세한 내용은 *utcoffset()*을 참조하십시오). 해당하면, DST 오프셋이 *utcoffset()*에서 반환된 UTC 오프셋에 이미 추가되어 있으므로, 따로 DST 정보를 얻는 데 관심이 없다면 *dst()*를 확인할 필요가 없습니다. 예를 들어, *datetime.timetuple()*은 *tzinfo* 어트리뷰트의 *dst()* 메서드를 호출하여 tm_isdst 플래그를 어떻게 설정할지를 결정하고, *tzinfo.fromutc()*는 시간대를 가로지를 때 DST 변경을 고려하기 위해 *dst()*를 호출합니다.

표준과 일광 절약 시간을 모두 모형화하는 *tzinfo* 서브 클래스의 인스턴스 *tz*는 다음과 같은 의미에서 일관되어야 합니다:

```
tz.utcoffset(dt) - tz.dst(dt)
```

는 dt.tzinfo == tz인 모든 *datetime* *dt*에 대해 같은 결과를 반환해야 합니다. 정상적인 *tzinfo* 서브 클래스에서, 이 표현식은 시간대의 "표준 오프셋"을 산출하는데, 이것은 날짜나 시간에 의존하지 않고, 지리적 위치에만 의존해야 합니다. *datetime.astimezone()* 구현은 이 일관성에 의존하지만, 위반을 감지할 수는 없습니다; 이를 보장하는 것은 프로그래머의 책임입니다. *tzinfo* 서브 클래스가 이를 보장 할 수 없으면, astimezone()와 상관없이 올바르게 작동하도록 *tzinfo.fromutc()*의 기본 구현을 재정의할 수 있습니다.

*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.  Then

    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*)

datetime 객체 *dt*에 해당하는 시간대 이름을 문자열로 반환합니다. 문자열 이름에 관한 어떤 것도 *datetime* 모듈에 의해 정의되지 않으며, 특별히 어떤 것을 의미해야 한다는 요구 사항이 없습니다. 예를 들어, "GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York"은 모두 유효한 응답입니다. 문자열 이름을 모르면 None을 반환합니다. 이것은 고정된 문자열이기보다 메서드인데, 주로 어떤 *tzinfo* 서브 클래스가 전달된 *dt*의 특정 값에 따라 다른 이름을 반환하기를 원하기 때문입니다. 특히 *tzinfo* 클래스가 일광 절약 시간을 고려할 때 그렇습니다.

*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.

datetime 메서드에 대한 응답으로 *datetime* 객체가 전달되면, dt.tzinfo는 *self*와 같은 객체입니다. 사용자 코드가 *tzinfo* 메서드를 직접 호출하지 않는 한, *tzinfo* 메서드는 이것에 의존할 수 있습니다. *tzinfo* 메서드가 *dt*를 지역 시간으로 해석하고, 다른 시간대의 객체를 걱정할 필요가 없도록 하려는 의도입니다.

서브 클래스가 재정의할 수 있는 *tzinfo* 메서드가 하나 더 있습니다:

tzinfo.**fromutc**(*dt*)

이것은 기본 *datetime.astimezone()* 구현에서 호출됩니다. 거기에서 호출되면, dt.tzinfo는 *self*이고, *dt*의 날짜와 시간 데이터는 UTC 시간으로 표시된 것으로 봅니다. *fromutc()*의 목적은 날짜와 시간 데이터를 조정하여, *self*의 지역 시간으로 동등한 datetime을 반환하는 것입니다.

대부분 *tzinfo* 서브 클래스는 문제없이 기본 *fromutc()* 구현을 상속할 수 있어야 합니다. 고정 오프셋 시간대와 표준과 일광 절약 시간을 모두 고려하는 시간대를, 해마다 DST 전환 시간이 다를 때도 일광 절약 시간을 처리할 수 있을 만큼 강력합니다. 기본 *fromutc()* 구현이 모든 경우에 올바르게 처리하지 못할 수 있는 시간대의 예는 (정치적 이유로 인해 발생할 수 있는) 특정 날짜와 시간에 따라 (UTC로부터의) 표준 오프셋이 달라지는 것입니다. 결과가 표준 오프셋이 변경되는 순간에 걸치는 시간 중 하나일 때, `astimezone()`과 *fromutc()*의 기본 구현은 여러분이 원하는 결과를 생성하지 못할 수 있습니다.

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)
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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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
#
# 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
# http://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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# 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

    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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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
  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())
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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
```

DST가 끝날 때 ("end" 줄), 잠재적으로 더 나쁜 문제가 있습니다: 지역 시간으로 명확하게 말할 수 없는 시(hour)가 있습니다: 일광 절약 시간의 마지막 한 시간. Eastern에서, 이것은 일광 절약 시간제가 끝나는 날의 5:MM UTC 형식의 시간입니다. 지역 벽시계는 1:59(일광 절약 시간)에서 다시 1:00(표준 시간)으로 도약합니다. 1:MM 형식의 지역 시간은 모호합니다. `astimezone()`은 두 개의 인접한 UTC 시(hour)를 같은 지역 시(hour)로 매핑하여 지역 시계 동작을 모방합니다. Eastern 예제에서, 5:MM과 6:MM 형식의 UTC 시간은 모두 Eastern으로 변환될 때 1:MM으로 매핑되지만, 이전 시간은 *fold* 어트리뷰트가 0으로 설정되고 이후 시간은 1로 설정됩니다. 예를 들어, 2016년 가을의 역 전환(back transition)에서, 다음과 같은 결과를 얻습니다

```
>>> 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
```

fold 어트리뷰트의 값만 다른 *datetime* 인스턴스는 비교에서 같다고 간주하는 것에 유의하십시오.

벽 시간 모호성을 견딜 수 없는 응용 프로그램은 명시적으로 *fold* 어트리뷰트 값을 확인하거나 하이브리드 *tzinfo* 서브 클래스를 사용하지 않아야 합니다; *timezone*이나 기타 고정 오프셋 *tzinfo* 서브 클래스(가령 오직 EST(고정 오프셋 -5시간)와 EDT(고정 오프셋 -4시간) 중 어느 한 가지만 나타내는 클래스)를 사용할 때는 모호함이 없습니다.

더 보기:

dateutil.tz 표준 라이브러리에는 UTC로부터의 임의의 고정 오프셋을 처리하기 위한 *timezone* 클래스와 UTC timezone 인스턴스로 *timezone.utc*가 있습니다.

dateutil.tz 라이브러리는 *IANA* 시간대 데이터베이스 (Olson 데이터베이스라고도 합니다)를 파이썬으로 가져옵니다. 사용을 권장합니다.

IANA timezone 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.

8.1.7 `timezone` 객체

timezone 클래스는 *tzinfo*의 서브 클래스이며, 각 인스턴스는 UTC로부터의 고정 오프셋으로 정의된 시간대를 나타냅니다. 이 클래스의 객체는 일 년 중 어떤 날에는 다른 오프셋이 사용되거나 민간 시간이 역사적으로 변해온 지역의 시간대 정보를 나타내는데 사용할 수 없음에 유의하십시오.

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.

버전 3.2에 추가.

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: `offset=timedelta(0)`에서 생성된 이름은 이제 'UTC+00:00'이 아니라 단순한 'UTC'입니다.

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**
UTC 시간대, `timezone(timedelta(0))`.

8.1.8 strftime()과 strptime() 동작

date, *datetime* 및 *time* 객체는 모두 `strftime(format)` 메서드를 지원하여, 명시적 포맷 문자열로 제어된 시간을 나타내는 문자열을 만듭니다. 대체로 말하자면, 모든 객체가 `timetuple()` 메서드를 지원하는 것은 아니지만, `d.strftime(fmt)`는 *time* 모듈의 `time.strftime(fmt, d.timetuple())`처럼 작동합니다.

반대로, *datetime.strptime()* 클래스 메서드는 날짜와 시간을 나타내는 문자열과 해당 포맷 문자열로 *datetime* 객체를 만듭니다. `datetime.strptime(date_string, format)`은 `format`에 초 미만의 성분이나 시간대 오프셋 정보가 포함된 경우를 제외하고는 `datetime(*(time.strptime(date_string, format)[0:6]))`과 동등합니다. 이것들은 `datetime.strptime`에서는 지원되지만 `time.strptime`에서는 버려집니다.

time 객체의 경우, time 객체에 해당 값이 없으므로, 연(year), 월(month) 및 일(day)의 포맷 코드는 사용하지 않아야 합니다. 어쨌든 사용되면, `1900`이 해당 연도로, `1`이 해당 월과 일로 대체됩니다.

date 객체의 경우, *date* 객체에 해당 값이 없으므로, 시(hour), 분(minute), 초(second) 및 마이크로초(microsecond)의 포맷 코드는 사용하지 않아야 합니다. 어쨌든 사용되면, `0`으로 대체됩니다.

datetime.strptime() 클래스 메서드의 경우, 기본값은 `1900-01-01T00:00:00.000`입니다: 포맷 문자열에 지정되지 않은 구성 요소는 기본값에서 가져옵니다.[2]

파이썬이 플랫폼 C 라이브러리의 `strftime()` 함수를 호출하고, 플랫폼 변형이 일반적이기 때문에, 지원되는 전체 포맷 코드 집합은 플랫폼에 따라 다릅니다. 여러분의 플랫폼에서 지원되는 모든 포맷 코드를 보려면, *strftime(3)* 설명서를 참조하십시오.

[2] `1900`이 윤년이 아니므로 `datetime.strptime('Feb 29', '%b %d')`를 전달하는 것은 실패합니다.

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.

다음은 C 표준(1989 버전)이 요구하는 모든 포맷 코드 목록이며, 표준 C 구현이 있는 모든 플랫폼에서 작동합니다. C 표준의 1999 버전에는 추가 형식 코드가 추가되었음에 유의하십시오.

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	마이크로초를 왼쪽에 0으로 채워진 십진수로.	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	시간대 이름 (객체가 나이브하면 빈 문자열).	(비어 있음), UTC, EST, CST	
%j	연중 일(day of the year)	001, 002, …, 366	(9)

C89 표준에서 요구하지 않는 몇 가지 추가 지시자가 편의상 포함되어 있습니다. 이 파라미터들은 모두 ISO 8601 날짜 값에 해당합니다. `strftime()` 메서드와 함께 사용될 때 모든 플랫폼에서 사용할 수 있는 것은 아닙니다. ISO 8601 연도와 ISO 8601 주 지시자는 위의 연도 및 주 번호 지시자와 교환할 수 없습니다. 불완전하거나 모호한 ISO 8601 지시자로 `strptime()`을 호출하면 *ValueError*가 발생합니다.

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)

버전 3.6에 추가: `%G`, `%u` 및 `%V`가 추가되었습니다.

노트:

(1) 포맷이 현재 로케일에 따라 다르므로, 출력값에 대해 가정을 할 때 주의해야 합니다. 필드 순서가 달라지며 (예를 들어, "월/일/년" 과 "일/월/년"), 출력에는 로케일의 기본 인코딩을 사용하여 인코딩된 유니코드 문자가 포함될 수 있습니다 (예를 들어, 현재 로케일이 `ja_JP`이면, 기본 인코딩은 `eucJP`, `SJIS` 또는 `utf-8` 중 하나일 수 있습니다; 현재 로케일의 인코딩을 결정하려면 *locale.getlocale()*을 사용하십시오).

(2) `strptime()` 메서드는 전체 [1, 9999] 범위에서 연도를 구문 분석할 수 있지만, 1000보다 작은 연도는 4자리 너비가 되도록 0으로 채워야 합니다.

버전 3.2에서 변경: 이전 버전에서 `strftime()` 메서드는 1900년 이상으로 제한되었습니다.

버전 3.3에서 변경: 버전 3.2에서, `strftime()` 메서드는 연도를 1000 이상으로 제한했습니다.

(3) `strptime()` 메서드와 함께 사용할 때, `%p` 지시자는 시간을 구문 분석하는 데 `%I` 지시문을 사용할 때만 출력 시간 필드에 영향을 줍니다.

(4) *time* 모듈과 달리, *datetime* 모듈은 윤초를 지원하지 않습니다.

(5) `strptime()` 메서드와 함께 사용할 때, `%f` 지시자는 하나에서 여섯 자리 숫자와 오른쪽의 0-채움을 받아들입니다. `%f`는 C 표준의 포맷 문자 집합에 대한 확장입니다 (하지만 datetime 객체에서 별도로 구현되므로 항상 사용할 수 있습니다).

(6) 나이브 객체의 경우, `%z` 와 `%Z` 포맷 코드는 빈 문자열로 치환됩니다.

어웨어 객체의 경우:

%z `utcoffset()`이 ±HHMM[SS[.ffffff]] 형식의 문자열로 변환됩니다. 여기서 HH는 UTC 오프셋 시간(hour)의 수를 나타내는 두 자리 숫자 문자열이고, MM은 UTC 오프셋 분의 수를 나타내는 두 자리 숫자 문자열이며, SS는 UTC 오프셋 초의 수를 나타내는 두 자리 숫자 문자열이며, ffffff는 UTC 오프셋 마이크로초의 수를 나타내는 6자리 숫자 문자열입니다. 오프셋이 딱 떨어지는 초면 ffffff 부분은 생략되며, offset이 딱 떨어지는 분이면 ffffff와 SS 부분이 모두 생략됩니다. 예를 들어, `utcoffset()`이 `timedelta(hours=-3, minutes=-30)`를 반환하면, `%z`는 `'-0330'` 문자열로 치환됩니다.

버전 3.7에서 변경: UTC 오프셋은 분 단위로 제한되지 않습니다.

버전 3.7에서 변경: `%z` 지시자가 `strptime()` 메서드에 제공될 때, UTC 오프셋에는 콜론이 시, 분 및 초 사이의 구분 기호로 사용될 수 있습니다. 예를 들어, `'+01:00:00'`은 1시간의 오프셋으로 구문 분석됩니다. 또한, `'Z'`를 제공하는 것은 `'+00:00'`과 같습니다.

%Z `tzname()`이 `None`을 반환하면, `%Z`는 빈 문자열로 치환됩니다. 그렇지 않으면 `%Z`는 문자열이어야 하는 반환 값으로 치환됩니다.

버전 3.2에서 변경: `%z` 지시자가 `strptime()` 메서드에 제공될 때, 어웨어 *datetime* 객체가 생성됩니다. 결과의 `tzinfo`는 *timezone* 인스턴스로 설정됩니다.

(7) `strptime()` 메서드와 함께 사용될 때, `%U` 와 `%W`는 요일과 달력 연도(`%Y`)가 지정되었을 때만 계산에 사용됩니다.

(8) `%U`와 `%W`와 비슷하게, `%V`는 요일과 ISO 연도(`%G`)가 `strptime()` 포맷 문자열에 지정되었을 때만 계산에 사용됩니다. 또한 `%G`와 `%Y`를 상호 교환할 수 없음에 유의하십시오.

(9) `strptime()` 메서드와 함께 사용할 때, 선행 0은 `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%J`, `%U`, `%W` 및 `%V` 포맷에서 선택적입니다. 포맷 `%y`에는 선행 0이 필요합니다.

8.2 `calendar` — General calendar-related functions

Source code: Lib/calendar.py

This module allows you to output calendars like the Unix **cal** program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use *setfirstweekday()* to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the *datetime* and *time* modules.

The functions and classes defined in this module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book "Calendrical Calculations", where it's the base calendar for all computations. Zero and negative years are interpreted as prescribed by the ISO 8601 standard. Year 0 is 1 BC, year -1 is 2 BC, and so on.

class calendar.**Calendar**(*firstweekday=0*)

Creates a *Calendar* object. *firstweekday* is an integer specifying the first day of the week. `0` is Monday (the default), `6` is Sunday.

A *Calendar* object provides several methods that can be used for preparing the calendar data for formatting. This class doesn't do any formatting itself. This is the job of subclasses.

Calendar instances have the following methods:

iterweekdays()

Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the *firstweekday* property.

itermonthdates(*year*, *month*)

Return an iterator for the month *month* (1–12) in the year *year*. This iterator will return all days (as *datetime.date* objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will simply be day of the month numbers. For the days outside of the specified month, the day number is `0`.

itermonthdays2(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a day of the month number and a week day number.

itermonthdays3(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.

버전 3.7에 추가.

itermonthdays4(*year*, *month*)
Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

버전 3.7에 추가.

monthdatescalendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven `datetime.date` objects.

monthdays2calendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

monthdayscalendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven day numbers.

yeardatescalendar(*year*, *width=3*)
Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to *width* months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are `datetime.date` objects.

yeardays2calendar(*year*, *width=3*)
Return the data for the specified year ready for formatting (similar to `yeardatescalendar()`). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

yeardayscalendar(*year*, *width=3*)
Return the data for the specified year ready for formatting (similar to `yeardatescalendar()`). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.**TextCalendar**(*firstweekday=0*)
This class can be used to generate plain text calendars.

`TextCalendar` instances have the following methods:

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)
Return a month's calendar in a multi-line string. If *w* is provided, it specifies the width of the date columns, which are centered. If *l* is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)
Print a month's calendar as returned by `formatmonth()`.

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
Return a *m*-column calendar for an entire year as a multi-line string. Optional parameters *w*, *l*, and *c* are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method. The earliest year for which a calendar can be generated is platform-dependent.

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
Print the calendar for an entire year as returned by `formatyear()`.

class calendar.**HTMLCalendar**(*firstweekday=0*)
This class can be used to generate HTML calendars.

`HTMLCalendar` instances have the following methods:

formatmonth(*theyear*, *themonth*, *withyear=True*)
Return a month's calendar as an HTML table. If *withyear* is true the year will be included in the header, otherwise just the month name will be used.

formatyear(*theyear*, *width=3*)
Return a year's calendar as an HTML table. *width* (defaulting to 3) specifies the number of months per row.

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)
Return a year's calendar as a complete HTML page. *width* (defaulting to 3) specifies the number of months per row. *css* is the name for the cascading style sheet to be used. `None` can be passed if no style sheet should be used. *encoding* specifies the encoding to be used for the output (defaulting to the system default encoding).

`HTMLCalendar` has the following attributes you can override to customize the CSS classes used by the calendar:

cssclasses
A list of CSS classes used for each weekday. The default class list is:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

more styles can be added for each day:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

Note that the length of this list must be seven items.

cssclass_noday
The CSS class for a weekday occurring in the previous or coming month.

버전 3.7에 추가.

cssclasses_weekday_head
A list of CSS classes used for weekday names in the header row. The default is the same as *cssclasses*.

버전 3.7에 추가.

cssclass_month_head
The month's head CSS class (used by `formatmonthname()`). The default value is `"month"`.

버전 3.7에 추가.

cssclass_month
The CSS class for the whole month's table (used by *formatmonth()*). The default value is `"month"`.

버전 3.7에 추가.

cssclass_year
The CSS class for the whole year's table of tables (used by *formatyear()*). The default value is `"year"`.

버전 3.7에 추가.

cssclass_year_head
The CSS class for the table head for the whole year (used by *formatyear()*). The default value is `"year"`.

버전 3.7에 추가.

Note that although the naming for the above described class attributes is singular (e.g. `cssclass_month` `cssclass_noday`), one can replace the single CSS class with a space separated list of CSS classes, for example:

```
"text-bold text-red"
```

Here is an example how `HTMLCalendar` can be customized:

```
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*)
This subclass of *TextCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)
This subclass of *HTMLCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

참고: The `formatweekday()` and `formatmonthname()` methods of these two classes temporarily change the current locale to the given *locale*. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

calendar.**setfirstweekday**(*weekday*)
Sets the weekday (`0` is Monday, `6` is Sunday) to start each week. The values `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are provided for convenience. For example, to set the first weekday to Sunday:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()
Returns the current setting for the weekday to start each week.

calendar.**isleap**(*year*)
Returns *True* if *year* is a leap year, otherwise *False*.

calendar.**leapdays**(*y1*, *y2*)
Returns the number of leap years in the range from *y1* to *y2* (exclusive), where *y1* and *y2* are years.

This function works for ranges spanning a century change.

calendar.**weekday**(*year*, *month*, *day*)
Returns the day of the week (`0` is Monday) for *year* (`1970`–…), *month* (`1`–`12`), *day* (`1`–`31`).

calendar.**weekheader**(*n*)
Return a header containing abbreviated weekday names. *n* specifies the width in characters for one weekday.

calendar.**monthrange**(*year*, *month*)
Returns weekday of first day of the month and number of days in month, for the specified *year* and *month*.

calendar.**monthcalendar**(*year*, *month*)
Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by *setfirstweekday()*.

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)
Prints a month's calendar as returned by *month()*.

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)
Returns a month's calendar in a multi-line string using the `formatmonth()` of the *TextCalendar* class.

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)
Prints the calendar for an entire year as returned by `calendar()`.

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)
Returns a 3-column calendar for an entire year as a multi-line string using the `formatyear()` of the `TextCalendar` class.

calendar.**timegm**(*tuple*)
An unrelated but handy function that takes a time tuple such as returned by the `gmtime()` function in the `time` module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, `time.gmtime()` and `timegm()` are each others' inverse.

The `calendar` module exports the following data attributes:

calendar.**day_name**
An array that represents the days of the week in the current locale.

calendar.**day_abbr**
An array that represents the abbreviated days of the week in the current locale.

calendar.**month_name**
An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

calendar.**month_abbr**
An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

더 보기:

Module ***datetime*** Object-oriented interface to dates and times with similar functionality to the `time` module.

Module ***time*** Low-level time related functions.

8.3 `collections` — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

Deprecated since version 3.3, will be removed in version 3.9: Moved *Collections 추상 베이스 클래스* to the `collections.abc` module. For backwards compatibility, they continue to be visible in this module through Python 3.8.

8.3.1 ChainMap objects

버전 3.3에 추가.

A *ChainMap* class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple *update()* calls.

The class can be used to simulate nested scopes and is useful in templating.

class collections.**ChainMap**(**maps*)

A *ChainMap* groups multiple dicts or other mappings together to create a single, updateable view. If no *maps* are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can be accessed or updated using the *maps* attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A *ChainMap* incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in *ChainMap*.

All of the usual dictionary methods are supported. In addition, there is a *maps* attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

maps

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

new_child(*m=None*)

Returns a new *ChainMap* containing a new map followed by all of the maps in the current instance. If `m` is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to `d.new_child()` is equivalent to: `ChainMap({}, *d.maps)`. This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

버전 3.4에서 변경: The optional `m` parameter was added.

parents

Property returning a new *ChainMap* containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the `nonlocal` keyword used in *nested scopes*. The use cases also parallel those for the built-in *super()* function. A reference to `d.parents` is equivalent to: `ChainMap(*d.maps[1:])`.

Note, the iteration order of a *ChainMap()* is determined by scanning the mappings last to first:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

This gives the same ordering as a series of *dict.update()* calls starting with the last mapping:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

더 보기:

- The MultiContext class in the Enthought CodeTools package has options to support writing to any mapping in the chain.
- Django's Context class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the *new_child()* method and the *parents* property.
- The Nested Contexts recipe has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A greatly simplified read-only version of Chainmap.

ChainMap Examples and Recipes

This section shows various approaches to working with chained maps.

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```
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'])
```

Example patterns for using the *ChainMap* class to simulate nested contexts:

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x'] = 1              # Set value in current context
d['x']                  # Get first key in the chain of contexts
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

The *ChainMap* class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    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'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.3.2 `Counter` objects

A counter tool is provided to support convenient and rapid tallies. For example:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> 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*])

A *Counter* is a *dict* subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The *Counter* class is similar to bags or multisets in other languages.

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```
>>> 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
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a *KeyError*:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

버전 3.1에 추가.

Counter objects support three methods beyond those available for all dictionaries:

elements()
Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element's count is less than one, *elements()* will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])
Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, *most_common()* returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract([*iterable-or-mapping*])
Elements are subtracted from an *iterable* or from another *mapping* (or counter). Like *dict.update()* but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
>>> 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})
```

버전 3.2에 추가.

The usual dictionary methods are available for *Counter* objects except for two which work differently for counters.

fromkeys(*iterable*)
This class method is not implemented for *Counter* objects.

update([*iterable-or-mapping*])
Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like *dict.update()* but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of `(key, value)` pairs.

Common patterns for working with *Counter* objects:

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
dict(c)                             # convert to a regular dictionary
c.items()                           # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))        # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]           # n least common elements
+c                                  # remove zero and negative counts
```

Several mathematical operations are provided for combining *Counter* objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                           # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                           # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                           # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                           # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

버전 3.3에 추가: Added support for unary plus, unary minus, and in-place multiset operations.

참고: Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The *Counter* class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you *could* store anything in the value field.
- The *most_common()* method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for *update()* and *subtract()* which allow negative and zero values for both inputs and outputs.
- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.
- The *elements()* method requires integer counts. It ignores zero and negative counts.

더 보기:

- Bag class in Smalltalk.
- Wikipedia entry for Multisets.
- C++ multisets tutorial with examples.

- For mathematical operations on multisets and their use cases, see *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*.
- To enumerate all distinct multisets of a given size over a given set of elements, see *itertools.combinations_with_replacement()*:

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.3.3 deque objects

class collections.**deque**([*iterable*[, *maxlen*]])

Returns a new deque object initialized left-to-right (using *append()*) with data from *iterable*. If *iterable* is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though *list* objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for `pop(0)` and `insert(0, v)` operations which change both the size and position of the underlying data representation.

If *maxlen* is not specified or is `None`, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the `tail` filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Deque objects support the following methods:

append(*x*)
Add *x* to the right side of the deque.

appendleft(*x*)
Add *x* to the left side of the deque.

clear()
Remove all elements from the deque leaving it with length 0.

copy()
Create a shallow copy of the deque.

버전 3.5에 추가.

count(*x*)
Count the number of deque elements equal to *x*.

버전 3.2에 추가.

extend(*iterable*)
Extend the right side of the deque by appending elements from the iterable argument.

extendleft(*iterable*)
Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

index(*x*[, *start*[, *stop*]])
Return the position of *x* in the deque (at or after index *start* and before index *stop*). Returns the first match or raises *ValueError* if not found.

버전 3.5에 추가.

insert(*i*, *x*)
Insert *x* into the deque at position *i*.

If the insertion would cause a bounded deque to grow beyond *maxlen*, an *IndexError* is raised.

버전 3.5에 추가.

pop()
Remove and return an element from the right side of the deque. If no elements are present, raises an *IndexError*.

popleft()
Remove and return an element from the left side of the deque. If no elements are present, raises an *IndexError*.

remove(*value*)
Remove the first occurrence of *value*. If not found, raises a *ValueError*.

reverse()
Reverse the elements of the deque in-place and then return `None`.

버전 3.2에 추가.

rotate(*n=1*)
Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left.

When the deque is not empty, rotating one step to the right is equivalent to `d.appendleft(d.pop())`, and rotating one step to the left is equivalent to `d.append(d.popleft())`.

Deque objects also provide one read-only attribute:

maxlen
Maximum size of a deque or `None` if unbounded.

버전 3.1에 추가.

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the `in` operator, and subscript references such as `d[-1]`. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

Starting in version 3.5, deques support `__add__()`, `__mul__()`, and `__imul__()`.

Example:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
['g', 'h', 'i']
>>> d[0]                            # peek at leftmost item
'g'
>>> d[-1]                           # peek at rightmost item
'i'

>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque Recipes

This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the `tail` filter in Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

A round-robin scheduler can be implemented with input iterators stored in a *deque*. Values are yielded from the active iterator in position zero. If that iterator is exhausted, it can be removed with *popleft()*; otherwise, it can be cycled back to the end with the *rotate()* method:

```
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:
            # Remove an exhausted iterator.
            iterators.popleft()
```

The *rotate()* method provides a way to implement *deque* slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement *deque* slicing, use a similar approach applying *rotate()* to bring a target element to the left side of the deque. Remove old entries with *popleft()*, add new entries with *extend()*, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

8.3.4 `defaultdict` objects

class collections.**defaultdict**([*default_factory*[, ...]])

Returns a new dictionary-like object. *defaultdict* is a subclass of the built-in *dict* class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the *dict* class and is not documented here.

The first argument provides the initial value for the *default_factory* attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the *dict* constructor, including keyword arguments.

defaultdict objects support the following method in addition to the standard *dict* operations:

__missing__(*key*)

If the *default_factory* attribute is `None`, this raises a *KeyError* exception with the *key* as argument.

If *default_factory* is not `None`, it is called without arguments to provide a default value for the given *key*, this value is inserted in the dictionary for the *key*, and returned.

If calling *default_factory* raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__()` method of the *dict* class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__()`.

Note that *__missing__()* is *not* called for any operations besides __getitem__(). This means that get() will, like normal dictionaries, return None as a default rather than using *default_factory*.

defaultdict objects support the following instance variable:

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.

defaultdict Examples

Using *list* as the *default_factory*, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> 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])]
```

Setting the *default_factory* to *int* makes the *defaultdict* useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the *default_factory* function calls *int()* to supply a default count of zero. The increment operation then builds up the count for each letter.

The function *int()* which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```
>>> 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>'
```

Setting the *default_factory* to *set* makes the *defaultdict* useful for building a dictionary of sets:

```
>>> 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})]
```

8.3.5 namedtuple() Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

collections.**namedtuple**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The *field_names* are a sequence of strings such as `['x', 'y']`. Alternatively, *field_names* can be a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a *keyword* such as *class*, *for*, *return*, *global*, *pass*, or *raise*.

If *rename* is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.

defaults can be `None` or an *iterable* of default values. Since fields with a default value must come after any fields without a default, the *defaults* are applied to the rightmost parameters. For example, if the fieldnames are `['x', 'y', 'z']` and the defaults are `(1, 2)`, then `x` will be a required argument, `y` will default to `1`, and `z` will default to `2`.

If *module* is defined, the `__module__` attribute of the named tuple is set to that value.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

버전 3.1에서 변경: Added support for *rename*.

버전 3.6에서 변경: The *verbose* and *rename* parameters became *keyword-only arguments*.

버전 3.6에서 변경: Added the *module* parameter.

버전 3.7에서 변경: Remove the *verbose* parameter and the `_source` attribute.

버전 3.7에서 변경: Added the *defaults* parameter and the `_field_defaults` attribute.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
(11, 22)
>>> p.x + p.y                # fields also accessible by name
33
>>> p                        # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the *csv* or *sqlite3* modules:

```
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)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

classmethod somenamedtuple.**_make**(*iterable*)
Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()
Return a new *dict* which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

버전 3.1에서 변경: Returns an *OrderedDict* instead of a regular *dict*.

somenamedtuple.**_replace**(***kwargs*)
Return a new instance of the named tuple replacing specified fields with new values:

```
>>> 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())
```

somenamedtuple.**_fields**
Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields            # view the field names
('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**
Dictionary mapping field names to default values.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

To retrieve a field whose name is stored in a string, use the *getattr()* function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the ** operator (as described in tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> 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
```

The subclass shown above sets __slots__ to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the *_fields* attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Docstrings can be customized by making direct assignments to the __doc__ fields:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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: Property docstrings became writeable.

Default values can be implemented by using *_replace()* to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
>>> janes_account = default_account._replace(owner='Jane')
```

더 보기:

- See *typing.NamedTuple* for a way to add type hints for named tuples. It also provides an elegant notation using the `class` keyword:

  ```
  class Component(NamedTuple):
      part_number: int
      weight: float
      description: Optional[str] = None
  ```

- See *types.SimpleNamespace()* for a mutable namespace based on an underlying dictionary instead of a tuple.
- The *dataclasses* module provides a decorator and functions for automatically adding generated special methods to user-defined classes.

8.3.6 `OrderedDict` objects

Ordered dictionaries are just like regular dictionaries but have some extra capabilities relating to ordering operations. They have become less important now that the built-in *dict* class gained the ability to remember insertion order (this new behavior became guaranteed in Python 3.7).

Some differences from *dict* still remain:

- The regular *dict* was designed to be very good at mapping operations. Tracking insertion order was secondary.
- The *OrderedDict* was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- Algorithmically, *OrderedDict* can handle frequent reordering operations better than *dict*. This makes it suitable for tracking recent accesses (for example in an LRU cache).
- The equality operation for *OrderedDict* checks for matching order.
- The `popitem()` method of *OrderedDict* has a different signature. It accepts an optional argument to specify which item is popped.
- *OrderedDict* has a `move_to_end()` method to efficiently reposition an element to an endpoint.
- Until Python 3.8, *dict* lacked a `__reversed__()` method.

class collections.**OrderedDict**([*items*])

Return an instance of a *dict* subclass that has methods specialized for rearranging dictionary order.

버전 3.1에 추가.

popitem(*last=True*)
The *popitem()* method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO (first-in, first-out) order if false.

move_to_end(*key*, *last=True*)
Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises *KeyError* if the *key* does not exist:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

버전 3.2에 추가.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using *reversed()*.

Equality tests between *OrderedDict* objects are order-sensitive and are implemented as `list(od1.items())==list(od2.items())`. Equality tests between *OrderedDict* objects and other *Mapping* objects are order-insensitive like regular dictionaries. This allows *OrderedDict* objects to be substituted anywhere a regular dictionary is used.

버전 3.5에서 변경: The items, keys, and values *views* of *OrderedDict* now support reverse iteration using *reversed()*.

버전 3.6에서 변경: With the acceptance of **PEP 468**, order is retained for keyword arguments passed to the *OrderedDict* constructor and its `update()` method.

OrderedDict Examples and Recipes

It is straightforward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        super().move_to_end(key)
```

An *OrderedDict* would also be useful for implementing variants of *functools.lru_cache()*:

```
class LRU(OrderedDict):
    'Limit size, evicting the least recently looked-up key when full'

    def __init__(self, maxsize=128, *args, **kwds):
        self.maxsize = maxsize
        super().__init__(*args, **kwds)

    def __getitem__(self, key):
        value = super().__getitem__(key)
        self.move_to_end(key)
        return value

    def __setitem__(self, key, value):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        super().__setitem__(key, value)
        if len(self) > self.maxsize:
            oldest = next(iter(self))
            del self[oldest]
```

8.3.7 UserDict objects

The class, *UserDict* acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from *dict*; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

class collections.**UserDict**([*initialdata*])

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the *data* attribute of *UserDict* instances. If *initialdata* is provided, *data* is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it be used for other purposes.

In addition to supporting the methods and operations of mappings, *UserDict* instances provide the following attribute:

data

A real dictionary used to store the contents of the *UserDict* class.

8.3.8 UserList objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from *list*; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.**UserList**([*list*])

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the *data* attribute of *UserList* instances. The instance's contents are initially set to a copy of *list*, defaulting to the empty list `[]`. *list* can be any iterable, for example a real Python list or a *UserList* object.

In addition to supporting the methods and operations of mutable sequences, *UserList* instances provide the following attribute:

data

A real *list* object used to store the contents of the *UserList* class.

Subclassing requirements: Subclasses of *UserList* are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

8.3.9 UserString objects

The class, *UserString* acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from *str*; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections.**UserString**(*seq*)

Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the *data* attribute of *UserString* instances. The instance's contents are initially set to a copy of *seq*. The *seq* argument can be any object which can be converted into a string using the built-in *str()* function.

In addition to supporting the methods and operations of strings, *UserString* instances provide the following attribute:

data

A real *str* object used to store the contents of the *UserString* class.

버전 3.5에서 변경: New methods `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, and `maketrans`.

8.4 collections.abc — 컨테이너의 추상 베이스 클래스

버전 3.3에 추가: 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 가능한지 또는 매핑인지입니다.

8.4.1 Collections 추상 베이스 클래스

collections 모듈은 다음과 같은 *ABC*를 제공합니다:

ABC	상속	추상 메서드	믹스인 메서드
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	*Iterable*	__next__	__iter__
Reversible	*Iterable*	__reversed__	
Generator	*Iterator*	send, throw	close, __iter__, __next__
Sized		__len__	
Callable		__call__	
Collection	*Sized*, *Iterable*, *Container*	__contains__, __iter__, __len__	
Sequence	*Reversible*, *Collection*	__getitem__, __len__	__contains__, __iter__, __reversed__, index 및 count
MutableSequence	*Sequence*	__getitem__, __setitem__, __delitem__, __len__, insert	상속된 *Sequence* 메서드와 append, reverse, extend, pop, remove 및 __iadd__
ByteString	*Sequence*	__getitem__, __len__	상속된 *Sequence* 메서드
Set	*Collection*	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __xor__ 및 isdisjoint
MutableSet	*Set*	__contains__, __iter__, __len__, add, discard	상속된 *Set* 메서드와 clear, pop, remove, __ior__, __iand__, __ixor__ 및 __isub__
Mapping	*Collection*	__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*		__len__
ItemsView	*MappingView*, *Set*		__contains__, __iter__
KeysView	*MappingView*, *Set*		__contains__, __iter__
ValuesView	*MappingView*, *Collection*		__contains__, __iter__
Awaitable		__await__	
Coroutine	*Awaitable*	send, throw	close
AsyncIterable		__aiter__	
AsyncIterator	*AsyncIterable*	__anext__	__aiter__
AsyncGenerator	*AsyncIterator*	asend, athrow	aclose, __aiter__, __anext__

class collections.abc.**Container**
class collections.abc.**Hashable**
class collections.abc.**Sized**
class collections.abc.**Callable**
각각 메서드 __contains__(), __hash__(), __len__() 및 __call__()을 제공하는 클래스의 ABC.

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.

버전 3.6에 추가.

class collections.abc.**Iterator**
*__iter__()*와 *__next__()* 메서드를 제공하는 클래스의 ABC. 이터레이터의 정의도 참조하십시오.

class collections.abc.**Reversible**
__reversed__() 메서드도 제공하는 이터러블 클래스의 ABC.

버전 3.6에 추가.

class collections.abc.**Generator**
send(),throw() 및 close() 메서드로 이터레이터를 확장하는 **PEP 342**에 정의된 프로토콜을 구현하는 제너레이터 클래스의 ABC. 제너레이터의 정의도 참조하십시오.

버전 3.5에 추가.

class collections.abc.**Sequence**
class collections.abc.**MutableSequence**
class collections.abc.**ByteString**
읽기 전용과 가변 시퀀스의 ABC.

구현 참고 사항: __iter__(), __reversed__() 및 index()와 같은 일부 믹스인(mixin) 메서드는 하부 __getitem__() 메서드를 반복적으로 호출합니다. 따라서, __getitem__()이 상수 액세스 속도로 구현되면 믹스인 메서드는 선형 성능을 갖습니다; 그러나 하부 메서드가 선형이면 (링크드 리스트에서처럼), 믹스인은 2차 함수 성능을 가지므로 재정의해야 할 수 있습니다.

버전 3.5에서 변경: index() 메서드는 *stop*과 *start* 인자에 대한 지원을 추가했습니다.

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()*이나 *asyncio.coroutine()*으로 데코레이트 된 제너레이터)은, __await__() 메서드가 없어도 어웨이터블 입니다. 이들에 대

해 `isinstance(gencoro, Awaitable)`를 사용하면 `False`가 반환됩니다. 이들을 감지하려면 *inspect.isawaitable()*을 사용하십시오.

버전 3.5에 추가.

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. 코루틴의 정의도 참조하십시오.

참고: CPython에서, 제너레이터 기반 코루틴(*types.coroutine()*이나 *asyncio.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.

버전 3.5에 추가.

class collections.abc.**AsyncIterable**
`__aiter__` 메서드를 제공하는 클래스의 ABC. 비동기 이터러블의 정의도 참조하십시오.

버전 3.5에 추가.

class collections.abc.**AsyncIterator**
`__aiter__`와 `__anext__` 메서드를 제공하는 클래스의 ABC. 비동기 이터레이터의 정의도 참조하십시오.

버전 3.5에 추가.

class collections.abc.**AsyncGenerator**
PEP 525와 **PEP 492**에 정의된 프로토콜을 구현하는 비동기 제너레이터 클래스의 ABC.

버전 3.6에 추가.

이러한 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):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

*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 가정합니다. 이 가정은 새로운 집합을 생성하기 위해 `cls(iterable)`를 호출하는 `_from_iterable()`이라는 내부 클래스 메서드로 분리되었습니다. *Set* 믹스인이 다른 생성자 서명을 갖는 클래스에서 사용되고 있으면, 이터러블 인자로부터 새 인스턴스를 생성할 수 있는 클래스 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.

8.5 heapq — Heap queue algorithm

Source code: Lib/heapq.py

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k*, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0]`.

The API below 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) Our pop method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two 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!

To create a heap, use a list initialized to `[]`, or you can transform a populated list into a heap via function *heapify()*.

The following functions are provided:

`heapq.`**`heappush`**(*heap*, *item*)
Push the value *item* onto the *heap*, maintaining the heap invariant.

heapq.**heappop**(*heap*)
Pop and return the smallest item from the *heap*, maintaining the 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*)
Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than `heappush()` followed by a separate call to `heappop()`.

heapq.**heapify**(*x*)
Transform list *x* into a heap, in-place, in linear time.

heapq.**heapreplace**(*heap*, *item*)
Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn't change. If the heap is empty, `IndexError` is raised.

This one step operation is more efficient than a `heappop()` followed by `heappush()` and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn't desired, consider using `heappushpop()` instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)
Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an *iterator* over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

Has two optional arguments which must be specified as keyword arguments.

key specifies a *key function* of one argument that is used to extract a comparison key from each input element. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed. To achieve behavior similar to `sorted(itertools.chain(*iterables), reverse=True)`, all iterables must be sorted from largest to smallest.

버전 3.5에서 변경: Added the optional *key* and *reverse* parameters.

heapq.**nlargest**(*n*, *iterable*, *key=None*)
Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`.

heapq.**nsmallest**(*n*, *iterable*, *key=None*)
Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key)[:n]`.

The latter two functions perform best for smaller values of *n*. For larger values, it is more efficient to use the `sorted()` function. Also, when `n==1`, it is more efficient to use the built-in `min()` and `max()` functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.

8.5.1 Basic Examples

A heapsort can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
>>> 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]
```

This is similar to `sorted(iterable)`, but unlike *`sorted()`*, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```
>>> 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')
```

8.5.2 Priority Queue Implementation Notes

A priority queue is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

Another solution to the problem of non-comparable tasks is to create a wrapper class that ignores the task item and only compares the priority field:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    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):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    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')
```

8.5.3 Theory

Heaps are arrays for which `a[k] <= a[2*k+1]` and `a[k] <= a[2*k+2]` for all *k*, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `a[0]` is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are *k*, not `a[k]`:

```
                                0

             1                                  2

      3              4                5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

In the tree above, each cell *k* is topping `2*k+1` and `2*k+2`. In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then

percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an O(n log n) sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing "runs" (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised[1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value "wins" over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a 'heap' module around. :-)

8.6 `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 일반적인 방법에 비해 개선된 것입니다. 이 모듈은 기본적인 이진 분할 알고리즘을 사용하기 때문에 *bisect*라고 불립니다. 소스 코드는 알고리즘의 실제 예로서 가장 유용할 수 있습니다 (경계 조건은 이미 옳습니다!).

다음과 같은 함수가 제공됩니다:

`bisect.`**`bisect_left`**(*a*, *x*, *lo=0*, *hi=len(a)*)

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 *i*는 배열 *a*를 이분하여, 왼쪽은 `all(val < x for val in a[lo:i])`, 오른쪽은 `all(val >= x for val in a[i:hi])`이 되도록 만듭니다.

`bisect.`**`bisect_right`**(*a*, *x*, *lo=0*, *hi=len(a)*)

[1] The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at "progressing" the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*)
*bisect_left()*와 비슷하지만, *a*에 있는 *x*의 기존 항목 뒤(오른쪽)에 오는 삽입 위치를 반환합니다.

반환된 삽입 위치 *i*는 배열 *a*를 이분하여, 왼쪽은 `all(val <= x for val in a[lo:i])`, 오른쪽은 `all(val > x for val in a[i:hi])`이 되도록 만듭니다.

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*)
*a*에 *x*를 정렬된 순서로 삽입합니다. *a*가 이미 정렬되었다고 가정할 때 `a.insert(bisect.bisect_left(a, x, lo, hi), x)`와 동등합니다. O(log n) 검색이 느린 O(n) 삽입 단계에 가려짐에 유념하십시오.

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*)
*insort_left()*와 비슷하지만, *a*에 *x*를 *x*의 기존 항목 다음에 삽입합니다.

더 보기:

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.

8.6.1 정렬된 리스트 검색하기

위의 *bisect()* 함수는 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):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        return a[i]
    raise ValueError
```

8.6.2 다른 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']
```

sorted() 함수와 달리, *bisect()* 함수는 *key* 나 *reversed* 인자를 갖는 것은 의미가 없는데, 비효율적인 설계 (연속적인 bisect 함수 호출이 이전의 모든 키 조회를 "기억"하지 못합니다)를 초래하기 때문입니다.

대신, 해당 레코드의 인덱스를 찾기 위해 미리 계산된 키 리스트를 검색하는 것이 좋습니다:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> 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)
```

8.7 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'	Py_UNICODE	유니코드 문자	2	(1)
'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) 'u' 형 코드는 파이썬의 구식 유니코드 문자(Py_UNICODE, 이것은 wchar_t입니다)에 해당합니다. 플랫폼에 따라, 16비트나 32비트가 될 수 있습니다.

'u'는 나머지 Py_UNICODE API와 함께 제거될 것입니다.

Deprecated since version 3.3, will be removed in version 4.0.

값의 실제 표현은 기계 아키텍처에 의해(엄격히 말하자면 C 구현에 의해) 결정됩니다. 실제 크기는 itemsize 어트리뷰트를 통해 액세스할 수 있습니다.

모듈은 다음 형을 정의합니다:

class array.**array**(*typecode*[, *initializer*])

항목이 *typecode*에 의해 제한되는 새 배열, 선택적인 *initializer* 값으로 초기화되는데, 리스트, 바이트열류 객체 또는 적절한 형의 요소에 대한 이터러블이어야 합니다.

리스트나 문자열이 주어지면, initializer는 새 배열의 *fromlist()*, *frombytes()* 또는 *fromunicode()* 메서드(아래를 참조하세요)에 전달되어 배열에 초기 항목을 추가합니다. 그렇지 않으면 이터러블 initializer가 *extend()* 메서드에 전달됩니다.

array.**typecodes**

사용 가능한 모든 형 코드가 있는 문자열.

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.

다음 데이터 항목과 메서드도 지원됩니다:

array.**typecode**

배열을 만드는 데 사용된 typecode 문자.

array.**itemsize**

내부 표현에서 하나의 배열 항목의 길이 (바이트).

array.**append**(*x*)

배열의 끝에 값 *x*로 새 항목을 추가합니다.

array.**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.itemsize로 계산할 수 있습니다. 이것은 특정 ioctl() 연산과 같

은 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.

`array.`**`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.

`array.`**`count`**`(`*x*`)`
배열 내에서 *x*가 등장하는 횟수를 반환합니다.

`array.`**`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.

`array.`**`frombytes`**`(`*s*`)`
문자열에서 항목을 추가합니다. 문자열을 기곗값(machine value)의 배열로 해석합니다 (마치 *fromfile()* 메서드를 사용하여 파일에서 읽은 것처럼).

버전 3.2에 추가: *fromstring()*은 명확하게 하려고 *frombytes()*로 이름을 바꿨습니다.

`array.`**`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. *f*는 실제 내장 파일 객체여야 합니다; `read()` 메서드를 가진 다른 것은 작동하지 않습니다.

`array.`**`fromlist`**`(`*list*`)`
리스트에서 항목을 추가합니다. 이것은 형 에러가 있으면 배열이 변경되지 않는다는 점만 제외하면 `for x in list: a.append(x)`와 동등합니다.

`array.`**`fromstring`**`()`
*frombytes()*의 폐지된 별칭.

Deprecated since version 3.2, will be removed in version 3.9.

`array.`**`fromunicode`**`(`*s*`)`
주어진 유니코드 문자열의 데이터로 이 배열을 확장합니다. 배열은 `'u'` 형의 배열이어야 합니다; 그렇지 않으면 *ValueError*가 발생합니다. 다른 형의 배열에 유니코드 데이터를 추가하려면 `array.frombytes(unicodestring.encode(enc))`를 사용하십시오.

`array.`**`index`**`(`*x*`)`
*i*가 배열에서 *x*가 처음 나타나는 인덱스가 되도록 가장 작은 *i*를 반환합니다.

`array.`**`insert`**`(`*i*`, `*x*`)`
i 위치 앞에 값이 *x*인 새 항목을 배열에 삽입합니다. 음수 값은 배열 끝에 상대적인 값으로 처리됩니다.

`array.`**`pop`**`(`[*i*]`)`
배열에서 인덱스 *i*에 있는 항목을 제거하고 이를 반환합니다. 선택적 인자의 기본값은 `-1`이므로, 기본적으로 마지막 항목이 제거되고 반환됩니다.

`array.`**`remove`**`(`*x*`)`
배열에서 첫 번째 *x*를 제거합니다.

array.**reverse**()
배열의 항목 순서를 뒤집습니다.

array.**tobytes**()
배열을 기곗값 배열로 변환하고 바이트열 표현(*tofile()* 메서드로 파일에 기록될 바이트 시퀀스와 같습니다)을 반환합니다.

버전 3.2에 추가: *tostring()*은 명확하게 하려고 *tobytes()*로 이름을 바꿨습니다.

array.**tofile**(*f*)
모든 항목을 (기곗값으로) 파일 객체 *f*에 씁니다.

array.**tolist**()
배열을 같은 항목이 있는 일반 리스트로 변환합니다.

array.**tostring**()
*tobytes()*의 폐지된 별칭.

Deprecated since version 3.2, will be removed in version 3.9.

array.**tounicode**()
배열을 유니코드 문자열로 변환합니다. 배열은 `'u'` 형의 배열이어야 합니다; 그렇지 않으면 *ValueError*가 발생합니다. 다른 형의 배열로부터 유니코드 문자열을 얻으려면 `array.tobytes().decode(enc)`를 사용하십시오.

배열 객체가 인쇄되거나 문자열로 변환될 때, `array(typecode, initializer)`로 표현됩니다. 배열이 비어 있으면 *initializer*가 생략되고, 그렇지 않으면 *typecode*가 `'u'` 인 경우 문자열이 되고, 그렇지 않으면 숫자 리스트가 됩니다. 문자열은 *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. 예:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

더 보기:

모듈 *struct* 이질적인(heterogeneous) 바이너리 데이터의 패킹과 언 패킹.

모듈 *xdrlib* 일부 원격 프로시저 호출 시스템에서 사용되는 XDR(External Data Representation) 데이터의 패킹과 언 패킹.

The Numerical Python Documentation Numeric Python 확장(NumPy)은 다른 배열형을 정의합니다; Numerical Python에 대한 더 자세한 정보는 http://www.numpy.org/ 를 참조하십시오.

8.8 weakref — Weak references

Source code: Lib/weakref.py

The *weakref* module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The *WeakKeyDictionary* and *WeakValueDictionary* classes supplied by the *weakref* module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a *WeakValueDictionary*, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

WeakKeyDictionary and *WeakValueDictionary* use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. *WeakSet* implements the *set* interface, but keeps weak references to its elements, just like a *WeakKeyDictionary* does.

finalize provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or *finalize* is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the *weakref* module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some *file objects*, *generators*, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

버전 3.2에서 변경: Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as *list* and *dict* do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

CPython implementation detail: Other built-in types such as *tuple* and *int* do not support weak references even when subclassed.

Extension types can easily be made to support weak references; see weakref-support.

class weakref.**ref**(*object*[, *callback*])

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause *None* to be returned. If *callback* is provided and not *None*, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's `__del__()` method.

Weak references are *hashable* if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If *hash()* is called the first time only after the *object* was deleted, the call will raise *TypeError*.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

__callback__
This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

버전 3.4에서 변경: Added the *__callback__* attribute.

`weakref.`**`proxy`**(*object*[, *callback*])
Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not *hashable* regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. *callback* is the same as the parameter of the same name to the *ref()* function.

`weakref.`**`getweakrefcount`**(*object*)
Return the number of weak references and proxies which refer to *object*.

`weakref.`**`getweakrefs`**(*object*)
Return a list of all weak reference and proxy objects which refer to *object*.

class `weakref.`**`WeakKeyDictionary`**([*dict*])
Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

WeakKeyDictionary objects have an additional method that exposes the internal references directly. The references are not guaranteed to be "live" at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

`WeakKeyDictionary.`**`keyrefs`**()
Return an iterable of the weak references to the keys.

class `weakref.`**`WeakValueDictionary`**([*dict*])
Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

WeakValueDictionary objects have an additional method that has the same issues as the `keyrefs()` method of *WeakKeyDictionary* objects.

`WeakValueDictionary.`**`valuerefs`**()
Return an iterable of the weak references to the values.

class `weakref.`**`WeakSet`**([*elements*])
Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

class `weakref.`**`WeakMethod`**(*method*)
A custom *ref* subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. *WeakMethod* has special code to recreate the bound method until either the object or the original function dies:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

버전 3.4에 추가.

class weakref.**finalize**(*obj*, *func*, **args*, ***kwargs*)

Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns *None*.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its *atexit* attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the *interpreter shutdown* when module globals are liable to have been replaced by *None*.

__call__()

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return *None*.

detach()

If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

peek()

If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

alive

Property which is true if the finalizer is alive, false otherwise.

atexit

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which *atexit* is true. They are called in reverse order of creation.

참고: It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method of *obj*.

버전 3.4에 추가.

weakref.**ReferenceType**
The type object for weak references objects.

weakref.**ProxyType**
The type object for proxies of objects which are not callable.

weakref.**CallableProxyType**
The type object for proxies of callable objects.

weakref.**ProxyTypes**
Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

더 보기:

PEP 205 - Weak References The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

8.8.1 Weak Reference Objects

Weak reference objects have no methods and no attributes besides *ref.__callback__*. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns *None*:

```
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for "liveness" creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of *ref* objects can be created through subclassing. This is used in the implementation of the *WeakValueDictionary* to reduce the memory overhead for each entry in the mapping. This may be most useful to

associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of *ref* can be used to store additional information about an object and affect the value that's returned when the referent is accessed:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.8.2 Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
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]
```

8.8.3 Finalizer Objects

The main benefit of using *finalize* is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> 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
```

You can unregister a finalizer using its *detach()* method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> 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
```

Unless you set the *atexit* attribute to *False*, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.8.4 Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs:

- the object is garbage collected,
- the object's `remove()` method is called, or
- the program exits.

We might try to implement the class using a `__del__()` method as follows:

```
class TempDir:
    def __init__(self):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during *interpreter shutdown*. So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```
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
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

참고: If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread `atexit.register()`, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

8.9 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.

8.9.1 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: ns`를 전달하는 것과 같은 효과가 있습니다.

버전 3.3에 추가.

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.

버전 3.3에 추가.

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 언패킹 된 결과로 대체된 튜플을 반환합니다. *bases* 항목이 *type*의 인스턴스이거나, `__mro_entries__` 메서드가 없으면, 반환 튜플에 변경되지 않은 상태로 포함됩니다.

버전 3.7에 추가.

더 보기:

PEP 560 - typing 모듈과 제네릭 형에 대한 코어 지원

8.9.2 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.

다음과 같은 형들에 대해 표준 이름이 정의됩니다:

types.**FunctionType**
types.**LambdaType**
사용자 정의 함수와 `lambda` 표현식이 만든 함수의 형.

types.**GeneratorType**
제너레이터 함수가 만든, 제너레이터-이터레이터 객체의 형.

types.**CoroutineType**
`async def` 함수가 만든 코루틴 객체의 형.

버전 3.5에 추가.

types.**AsyncGeneratorType**
비동기 제너레이터 함수가 만든, 비동기 제너레이터-이터레이터 객체의 형.

버전 3.6에 추가.

types.**CodeType**
*compile()*이 반환하는 것과 같은 코드 객체의 형.

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.

버전 3.7에 추가.

types.**MethodWrapperType**
일부 내장 데이터형과 베이스 클래스의 연결된*(bound)* 메서드의 형. 예를 들어 `object().__str__`의 형입니다.

버전 3.7에 추가.

types.**MethodDescriptorType**
*str.join()*과 같은 일부 내장 데이터형의 메서드의 형.

버전 3.7에 추가.

types.**ClassMethodDescriptorType**
`dict.__dict__['fromkeys']`와 같은 일부 내장 데이터형의 연결되지 않은*(unbound)* 클래스 메서드의 형.

버전 3.7에 추가.

class types.**ModuleType**(*name*, *doc*=*None*)
모듈의 형. 생성자는 만들 모듈의 이름과 선택적으로 독스트링을 취합니다.

참고: 다양한 임포트 제어 어트리뷰트를 설정하려면 *importlib.util.module_from_spec()*을 사용하여 새 모듈을 만드십시오.

__doc__
모듈의 독스트링. 기본값은 None.

__loader__
모듈을 로드한 로더. 기본값은 None.

버전 3.4에서 변경: 기본값은 None. 이전에는 어트리뷰트가 선택적이었습니다.

__name__
모듈의 이름

__package__
모듈이 속한 패키지. 모듈이 최상위 수준이면 (즉, 특정 패키지의 일부가 아니면) 어트리뷰트를 ''로 설정해야 하며, 그렇지 않으면 패키지 이름으로 설정해야 합니다 (모듈이 패키지 자체이면 __name__ 일 수 있습니다). 기본값은 None.

버전 3.4에서 변경: 기본값은 None. 이전에는 어트리뷰트가 선택적이었습니다.

class types.**TracebackType**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)
sys.exc_info()[2]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.

사용 가능한 어트리뷰트와 연산에 대한 자세한 내용은 언어 레퍼런스를 참조하십시오.

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.

CPython implementation detail: 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.

버전 3.3에 추가.

key in proxy
하부 매핑에 키 *key*가 있으면 True를, 그렇지 않으면 False를 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 None으로 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.

8.9.3 추가 유틸리티 클래스와 함수

class types.**SimpleNamespace**
이름 공간에 대한 어트리뷰트 액세스와 의미 있는 repr을 제공하는 간단한 *object* 서브 클래스.

*object*와 달리, SimpleNamespace를 사용하면 어트리뷰트를 추가하고 제거할 수 있습니다. SimpleNamespace 객체가 키워드 인자로 초기화되면, 하부 이름 공간에 직접 추가됩니다.

형은 다음 코드와 대략 동등합니다:

```
class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        return self.__dict__ == other.__dict__
```

SimpleNamespace는 class NS: pass의 대체품으로 유용할 수 있습니다. 하지만, 구조화된 레코드형에는 *namedtuple()*을 대신 사용하십시오.

버전 3.3에 추가.

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 정의하는 데 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 Enum을 참조하십시오).

버전 3.4에 추가.

8.9.4 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.

버전 3.5에 추가.

8.10 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**(*x*)
*x*의 얕은 사본을 반환합니다.

copy.**deepcopy**(*x*[, *memo*])
*x*의 깊은 사본을 반환합니다.

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, 윈도우, 배열과 같은 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. 전자는 얕은 복사 연산을 실행하기 위해 호출됩니다; 추가적인 인자를 전달 하지 않습니다. 후자는 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.

더 보기:

모듈 *pickle* 객체 상태 조회와 복원을 지원하는데 사용되는 특수 메서드에 관한 논의

8.11 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. 너비 제한을 조정해야 하면 *PrettyPrinter* 객체를 명시적으로 만드십시오.

딕셔너리는 디스플레이를 계산하기 전에 키로 정렬됩니다.

pprint 모듈은 하나의 클래스를 정의합니다:

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*)

PrettyPrinter 인스턴스를 만듭니다. 이 생성자는 여러 키워드 매개 변수를 인식합니다. 출력 스트림은 *stream* 키워드를 사용하여 설정할 수 있습니다; 스트림 객체에서 사용되는 유일한 메서드는 파일 프로토콜의 `write()` 메서드입니다. 지정하지 않으면, *PrettyPrinter*는 `sys.stdout`을 사용합니다. 각 재귀 수준에 대해 들여쓰기하는 양은 *indent*로 지정합니다; 기본값은 1 입니다. 다른 값은 출력이 약간 이상하게 보일 수 있지만, 중첩을 쉽게 알아낼 수 있습니다. 인쇄될 수 있는 수준의 수는 *depth*로 제어합니다; 인쇄 중인 데이터 구조가 너무 깊으면, 다음에 포함된 수준은 `...`로 대체됩니다. 기본적으로, 포맷되는 객체의 깊이에는 제한이 없습니다. 원하는 출력 폭은 *width* 매개 변수를 사용하여 제한합니다; 기본값은 80자입니다. 제한된 너비 내에서 구조를 포맷할 수 없으면, 최선의 노력을 기울입니다. *compact*가 거짓 (기본값) 이면, 긴 시퀀스의 각 항목이 별도의 줄로 포맷됩니다. *compact*가 참이면 *width* 내에 들어갈 수 있는 최대한 많은 항목을 각 출력할 줄에 포맷합니다.

버전 3.4에서 변경: *compact* 매개 변수가 추가되었습니다.

```
>>> 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', (...)))))))
```

pprint 모듈은 몇 가지 단축 함수도 제공합니다:

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)
*object*의 포맷된 표현을 문자열로 반환합니다. *indent*, *width*, *depth* 및 *compact*는 포매팅 매개 변수로 *PrettyPrinter* 생성자에 전달됩니다.

버전 3.4에서 변경: *compact* 매개 변수가 추가되었습니다.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)
*object*의 포맷된 표현에 줄 바꿈을 추가해서 *stream*에 인쇄합니다. *stream*이 `None`이면, `sys.stdout`이 사용됩니다. 이것은 *print()* 함수 대신 대화형 인터프리터에서 값을 검사하는 데 사용할 수 있습니다 (스크프 내에서 사용하기 위해 `print = pprint.pprint`를 다시 대입할 수도 있습니다). *indent*, *width*, *depth* 및 *compact*는 포매팅 매개 변수로 *PrettyPrinter* 생성자에 전달됩니다.

버전 3.4에서 변경: *compact* 매개 변수가 추가되었습니다.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

pprint.**isreadable**(*object*)
Determine if the formatted representation of *object* is “readable”, or can be used to reconstruct the value using *eval()*. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)
*object*가 재귀적 표현을 요구하는지 판단합니다.

또 하나의 지원 함수가 정의됩니다:

pprint.**saferepr**(*object*)
재귀적 데이터 구조에 대해 보호되는, *object*의 문자열 표현을 반환합니다. *object*의 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']"
```

8.11.1 PrettyPrinter 객체

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.

8.11.2 예제

pprint() 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/json') as resp:
...     project_info = json.load(resp)['info']
```

기본적인 형태에서, *pprint()*는 전체 객체를 보여줍니다:

```
>>> pprint.pprint(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/',
 '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.pprint(project_info, depth=1)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
{'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.pprint(project_info, depth=1, width=60)
{'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 '
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                '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'}
```

8.12 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**
내장 *repr()*과 유사한 함수를 구현하는 데 유용한 포매팅 서비스를 제공하는 클래스; 과도하게 긴 표현의 생성을 피하고자 객체 형별로 크기 제한이 추가됩니다.

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*)
aRepr의 *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'>
```

버전 3.2에 추가.

8.12.1 Repr 객체

Repr 인스턴스는 여러 객체 형의 표현에 대한 크기 제한과 특정 객체 형을 포맷하는 메서드를 제공하는데 사용될 수 있습니다.

`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.`**`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.

8.12.2 Repr 객체 서브클래싱

*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))          # prints '<stdin>'
```

8.13 enum — Support for enumerations

버전 3.4에 추가.

Source code: Lib/enum.py

An enumeration is a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over.

8.13.1 Module Contents

This module defines four enumeration classes that can be used to define unique sets of names and values: *Enum*, *IntEnum*, *Flag*, and *IntFlag*. It also defines one decorator, *unique()*, and one helper, *auto*.

class enum.**Enum**
Base class for creating enumerated constants. See section *Functional API* for an alternate construction syntax.

class enum.**IntEnum**
Base class for creating enumerated constants that are also subclasses of *int*.

class enum.**IntFlag**
Base class for creating enumerated constants that can be combined using the bitwise operators without losing their *IntFlag* membership. *IntFlag* members are also subclasses of *int*.

class enum.**Flag**
Base class for creating enumerated constants that can be combined using the bitwise operations without losing their *Flag* membership.

enum.**unique**()
Enum class decorator that ensures only one name is bound to any one value.

class enum.**auto**
Instances are replaced with an appropriate value for Enum members. Initial value starts at 1.

버전 3.6에 추가: `Flag`, `IntFlag`, `auto`

8.13.2 Creating an Enum

Enumerations are created using the `class` syntax, which makes them easy to read and write. An alternative creation method is described in *Functional API*. To define an enumeration, subclass *Enum* as follows:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...
```

참고: Enum member values

Member values can be anything: *int*, *str*, etc.. If the exact value is unimportant you may use *auto* instances and an appropriate value will be chosen for you. Care must be taken if you mix *auto* with other values.

참고: Nomenclature

- The class `Color` is an *enumeration* (or *enum*)
- The attributes `Color.RED`, `Color.GREEN`, etc., are *enumeration members* (or *enum members*) and are functionally constants.
- The enum members have *names* and *values* (the name of `Color.RED` is `RED`, the value of `Color.BLUE` is `3`, etc.)

참고: Even though we use the `class` syntax to create Enums, Enums are not normal Python classes. See *How are Enums different?* for more details.

Enumeration members have human readable string representations:

```
>>> print(Color.RED)
Color.RED
```

…while their `repr` has more information:

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

The *type* of an enumeration member is the enumeration it belongs to:

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>>
```

Enum members also have a property that contains just their item name:

```
>>> print(Color.RED.name)
RED
```

Enumerations support iteration, in definition order:

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

Enumeration members are hashable, so they can be used in dictionaries and sets:

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3 Programmatic access to enumeration members and their attributes

Sometimes it's useful to access members in enumerations programmatically (i.e. situations where `Color.RED` won't do because the exact color is not known at program-writing time). `Enum` allows such access:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

If you want to access enum members by *name*, use item access:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

If you have an enum member and need its `name` or `value`:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4 Duplicating enum members and values

Having two enum members with the same name is invalid:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

However, two enum members are allowed to have the same value. Given two members A and B with the same value (and A defined first), B is an alias to A. By-value lookup of the value of A and B will return A. By-name lookup of B will also return A:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

참고: Attempting to create a member with the same name as an already defined attribute (another member, a method, etc.) or attempting to create an attribute with the same name as a member is not allowed.

8.13.5 Ensuring unique enumeration values

By default, enumerations allow multiple names as aliases for the same value. When this behavior isn't desired, the following decorator can be used to ensure each value is used only once in the enumeration:

`@enum.unique`

A `class` decorator specifically for enumerations. It searches an enumeration's `__members__` gathering any aliases it finds; if any are found *`ValueError`* is raised with the details:

```
>>> 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
```

8.13.6 Using automatic values

If the exact value is unimportant you can use *auto*:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

The values are chosen by `_generate_next_value_()`, which can be overridden:

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.
↪WEST: 'WEST'>]
```

참고: The goal of the default `_generate_next_value_()` methods is to provide the next *int* in sequence with the last *int* provided, but the way it does this is an implementation detail and may change.

참고: The `_generate_next_value_()` method must be defined before any members.

8.13.7 Iteration

Iterating over the members of an enum does not provide the aliases:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

The special attribute `__members__` is an ordered dictionary mapping names to members. It includes all names defined in the enumeration, including the aliases:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

The __members__ attribute can be used for detailed programmatic access to the enumeration members. For example, finding all the aliases:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8 Comparisons

Enumeration members are compared by identity:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Ordered comparisons between enumeration values are *not* supported. Enum members are not integers (but see *IntEnum* below):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

Equality comparisons are defined though:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Comparisons against non-enumeration values will always compare not equal (again, *IntEnum* was explicitly designed to behave differently, see below):

```
>>> Color.BLUE == 2
False
```

8.13.9 Allowed members and attributes of enumerations

The examples above use integers for enumeration values. Using integers is short and handy (and provided by default by the *Functional API*), but not strictly enforced. In the vast majority of use-cases, one doesn't care what the actual value of an enumeration is. But if the value *is* important, enumerations can have arbitrary values.

Enumerations are Python classes, and can have methods and special methods as usual. If we have this enumeration:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

Then:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

The rules for what is allowed are as follows: names that start and end with a single underscore are reserved by enum and cannot be used; all other attributes defined within an enumeration will become members of this enumeration, with the exception of special methods (`__str__()`, `__add__()`, etc.), descriptors (methods are also descriptors), and variable names listed in `_ignore_`.

Note: if your enumeration defines `__new__()` and/or `__init__()` then whatever value(s) were given to the enum member will be passed into those methods. See *Planet* for an example.

8.13.10 Restricted Enum subclassing

A new *Enum* class must have one base Enum class, up to one concrete data type, and as many *object*-based mixin classes as needed. The order of these base classes is:

```
class EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

Also, subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

But this is allowed:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...
```

Allowing subclassing of enums that define members would lead to a violation of some important invariants of types and instances. On the other hand, it makes sense to allow sharing some common behavior between a group of enumerations. (See *OrderedEnum* for an example.)

8.13.11 Pickling

Enumerations can be pickled and unpickled:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

The usual restrictions for pickling apply: picklable enums must be defined in the top level of a module, since unpickling requires them to be importable from that module.

참고: With pickle protocol version 4 it is possible to easily pickle enums nested in other classes.

It is possible to modify how Enum members are pickled/unpickled by defining *__reduce_ex__()* in the enumeration class.

8.13.12 Functional API

The *Enum* class is callable, providing the following functional API:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

The semantics of this API resemble *namedtuple*. The first argument of the call to *Enum* is the name of the enumeration.

The second argument is the *source* of enumeration member names. It can be a whitespace-separated string of names, a sequence of names, a sequence of 2-tuples with key/value pairs, or a mapping (e.g. dictionary) of names to values. The last two options enable assigning arbitrary values to enumerations; the others auto-assign increasing integers starting with 1 (use the `start` parameter to specify a different starting value). A new class derived from *Enum* is returned. In other words, the above assignment to `Animal` is equivalent to:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

The reason for defaulting to `1` as the starting number and not `0` is that `0` is `False` in a boolean sense, but enum members all evaluate to `True`.

Pickling enums created with the functional API can be tricky as frame stack implementation details are used to try and figure out which module the enumeration is being created in (e.g. it will fail if you use a utility function in separate module, and also may not work on IronPython or Jython). The solution is to specify the module name explicitly as follows:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

경고: If `module` is not supplied, and Enum cannot determine what it is, the new Enum members will not be unpicklable; to keep errors closer to the source, pickling will be disabled.

The new pickle protocol 4 also, in some circumstances, relies on *__qualname__* being set to the location where pickle will be able to find the class. For example, if the class was made available in class SomeData in the global scope:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

The complete signature is:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-
↪in class>, start=1)
```

value What the new Enum class will record as its name.

names The Enum members. This can be a whitespace or comma separated string (values will start at 1 unless otherwise specified):

```
'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'
```

or an iterator of names:

```
['RED', 'GREEN', 'BLUE']
```

or an iterator of (name, value) pairs:

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

or a mapping:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

module name of module where new Enum class can be found.

qualname where in module new Enum class can be found.

type type to mix in to new Enum class.

start number to start counting at if only names are passed in.

버전 3.5에서 변경: The *start* parameter was added.

8.13.13 Derived Enumerations

IntEnum

The first variation of *Enum* that is provided is also a subclass of *int*. Members of an *IntEnum* can be compared to integers; by extension, integer enumerations of different types can also be compared to each other:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

However, they still can't be compared to standard *Enum* enumerations:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum values behave like integers in other ways you'd expect:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

The next variation of *Enum* provided, *IntFlag*, is also based on *int*. The difference being *IntFlag* members can be combined using the bitwise operators (&, |, ^, ~) and the result is still an *IntFlag* member. However, as the name implies, *IntFlag* members also subclass *int* and can be used wherever an *int* is used. Any operation on an *IntFlag* member besides the bit-wise operations will lose the *IntFlag* membership.

버전 3.6에 추가.

Sample *IntFlag* class:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

It is also possible to name the combinations:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

Another important difference between *IntFlag* and *Enum* is that if no flags are set (the value is 0), its boolean evaluation is *False*:

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

Because *IntFlag* members are also subclasses of *int* they can be combined with them:

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

Flag

The last variation is *Flag*. Like *IntFlag*, *Flag* members can be combined using the bitwise operators (&, |, ^, ~). Unlike *IntFlag*, they cannot be combined with, nor compared against, any other *Flag* enumeration, nor *int*. While it is possible to specify the values directly it is recommended to use *auto* as the value and let *Flag* select an appropriate value.

버전 3.6에 추가.

Like *IntFlag*, if a combination of *Flag* members results in no flags being set, the boolean evaluation is *False*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Individual flags should have values that are powers of two (1, 2, 4, 8, ⋯), while combinations of flags won't:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

Giving a name to the "no flags set" condition does not change its boolean value:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

참고: For the majority of new code, *Enum* and *Flag* are strongly recommended, since *IntEnum* and *IntFlag* break some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). *IntEnum* and *IntFlag* should be used only in cases where *Enum* and *Flag* will not do; for example, when integer constants are replaced with enumerations, or for interoperability with other systems.

Others

While *IntEnum* is part of the *enum* module, it would be very simple to implement independently:

```
class IntEnum(int, Enum):
    pass
```

This demonstrates how similar derived enumerations can be defined; for example a `StrEnum` that mixes in *str* instead of *int*.

Some rules:

1. When subclassing *Enum*, mix-in types must appear before *Enum* itself in the sequence of bases, as in the *IntEnum* example above.
2. While *Enum* can have members of any type, once you mix in an additional type, all the members must have values of that type, e.g. *int* above. This restriction does not apply to mix-ins which only add methods and don't specify another data type such as *int* or *str*.
3. When another data type is mixed in, the `value` attribute is *not the same* as the enum member itself, although it is equivalent and will compare equal.

4. %-style formatting: *%s* and *%r* call the *Enum* class's `__str__()` and `__repr__()` respectively; other codes (such as *%i* or *%h* for IntEnum) treat the enum member as its mixed-in type.
5. Formatted string literals, *str.format()*, and *format()* will use the mixed-in type's `__format__()`. If the *Enum* class's *str()* or *repr()* is desired, use the *!s* or *!r* format codes.

8.13.14 Interesting examples

While *Enum*, *IntEnum*, *IntFlag*, and *Flag* are expected to cover the majority of use-cases, they cannot cover them all. Here are recipes for some different types of enumerations that can be used directly, or as examples for creating one's own.

Omitting values

In many use-cases one doesn't care what the actual value of an enumeration is. There are several ways to define this type of simple enumeration:

- use instances of *auto* for the value
- use instances of *object* as the value
- use a descriptive string as the value
- use a tuple as the value and a custom `__new__()` to replace the tuple with an *int* value

Using any of these methods signifies to the user that these values are not important, and also enables one to add, remove, or reorder members without having to renumber the remaining members.

Whichever method you choose, you should provide a *repr()* that also hides the (unimportant) value:

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

Using `auto`

Using *auto* would look like:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

Using `object`

Using *`object`* would look like:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

Using a descriptive string

Using a string as the value would look like:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

Using a custom `__new__()`

Using an auto-numbering `__new__()` would look like:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

참고: The `__new__()` method, if defined, is used during creation of the Enum members; it is then replaced by Enum's `__new__()` which is used after class creation for lookup of existing members.

OrderedEnum

An ordered enumeration that is not based on *IntEnum* and so maintains the normal *Enum* invariants (such as not being comparable to other enumerations):

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

Raises an error if a duplicate member name is found instead of creating an alias:

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:  %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

참고: This is a useful example for subclassing Enum to add or change other behaviors as well as disallowing aliases. If the only desired change is disallowing aliases, the *unique()* decorator can be used instead.

Planet

If `__new__()` or `__init__()` is defined the value of the enum member will be passed to those methods:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

TimePeriod

An example to show the `_ignore_` attribute in use:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelta(days=1)>]
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.
↪timedelta(days=366)>]
```

8.13.15 How are Enums different?

Enums have a custom metaclass that affects many aspects of both derived Enum classes and their instances (members).

Enum Classes

The `EnumMeta` metaclass is responsible for providing the `__contains__()`, `__dir__()`, `__iter__()` and other methods that allow one to do things with an *Enum* class that fail on a typical class, such as *list(Color)* or *some_enum_var in Color*. `EnumMeta` is responsible for ensuring that various other methods on the final *Enum* class are correct (such as `__new__()`, *__getnewargs__()*, `__str__()` and `__repr__()`).

Enum Members (aka instances)

The most interesting thing about Enum members is that they are singletons. `EnumMeta` creates them all while it is creating the *Enum* class itself, and then puts a custom `__new__()` in place to ensure that no new ones are ever instantiated by returning only the existing member instances.

Finer Points

Supported `__dunder__` names

`__members__` is an `OrderedDict` of `member_name:member` items. It is only available on the class.

`__new__()`, if specified, must create and return the enum members; it is also a very good idea to set the member's `_value_` appropriately. Once all the members are created it is no longer used.

Supported `_sunder_` names

- `_name_` – name of the member
- `_value_` – value of the member; can be set / modified in `__new__`
- `_missing_` – a lookup function used when a value is not found; may be overridden
- `_ignore_` – a list of names, either as a *list()* or a *str()*, that will not be transformed into members, and will be removed from the final class
- `_order_` – used in Python 2/3 code to ensure member order is consistent (class attribute, removed during class creation)
- `_generate_next_value_` – used by the *Functional API* and by `auto` to get an appropriate value for an enum member; may be overridden

버전 3.6에 추가: `_missing_`, `_order_`, `_generate_next_value_`

버전 3.7에 추가: `_ignore_`

To help keep Python 2 / Python 3 code in sync an `_order_` attribute can be provided. It will be checked against the actual order of the enumeration and raise an error if the two do not match:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

참고: In Python 2 code the `_order_` attribute is necessary as definition order is lost before it can be recorded.

`Enum` member type

`Enum` members are instances of their *`Enum`* class, and are normally accessed as `EnumClass.member`. Under certain circumstances they can also be accessed as `EnumClass.member.member`, but you should never do this as that lookup may fail or, worse, return something besides the *`Enum`* member you are looking for (this is another good reason to use all-uppercase names for members):

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

버전 3.5에서 변경.

Boolean value of `Enum` classes and members

`Enum` members that are mixed with non-*`Enum`* types (such as *`int`*, *`str`*, etc.) are evaluated according to the mixed-in type's rules; otherwise, all members evaluate as *`True`*. To make your own Enum's boolean evaluation depend on the member's value add the following to your class:

```
def __bool__(self):
    return bool(self.value)
```

`Enum` classes always evaluate as *`True`*.

`Enum` classes with methods

If you give your *`Enum`* subclass extra methods, like the *Planet* class above, those methods will show up in a *`dir()`* of the member, but not of the class:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__
↪class__', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

Combining members of `Flag`

If a combination of Flag members is not named, the *repr()* will include all named flags and all named combinations of flags that are in the value:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

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()
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. 이 프로퍼티는 기약 분수의 값이어야 합니다. 또한 *float()* 함수를 위한 기본값으로 사용됩니다.

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. `**` 를 위한 메서드와 비트 연산 `<<`, `>>`, `&`, `^`, `|`, `~` 를 추가합니다.

9.1.2 형 구현을 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:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        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):
        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:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.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` — Mathematical functions

This module provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the *cmath* module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

9.2.1 Number-theoretic and representation functions

math.**ceil**(*x*)
Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__()`, which should return an *Integral* value.

math.**copysign**(*x*, *y*)
Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns *-1.0*.

math.**fabs**(*x*)
Return the absolute value of *x*.

math.**factorial**(*x*)
Return *x* factorial as an integer. Raises *ValueError* if *x* is not integral or is negative.

math.**floor**(*x*)
Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__()`, which should return an *Integral* value.

math.**fmod**(*x*, *y*)
Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer *n* such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function *fmod()* is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

math.**frexp**(*x*)
Return the mantissa and exponent of *x* as the pair `(m, e)`. *m* is a float and *e* is an integer such that `x == m * 2**e` exactly. If *x* is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to "pick apart" the internal representation of a float in a portable way.

math.**fsum**(*iterable*)
Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

The algorithm's accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating point summation.

math.**gcd**(*a*, *b*)
Return the greatest common divisor of the integers *a* and *b*. If either *a* or *b* is nonzero, then the value of `gcd(a, b)` is the largest positive integer that divides both *a* and *b*. `gcd(0, 0)` returns `0`.

버전 3.5에 추가.

math.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)
Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be greater than zero.

abs_tol is the minimum absolute tolerance – useful for comparisons near zero. *abs_tol* must be at least zero.

If no errors occur, the result will be: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

버전 3.5에 추가.

더 보기:

PEP 485 – A function for testing approximate equality

`math.`**`isfinite`**(*x*)
Return `True` if *x* is neither an infinity nor a NaN, and `False` otherwise. (Note that `0.0` *is* considered finite.)

버전 3.2에 추가.

`math.`**`isinf`**(*x*)
Return `True` if *x* is a positive or negative infinity, and `False` otherwise.

`math.`**`isnan`**(*x*)
Return `True` if *x* is a NaN (not a number), and `False` otherwise.

`math.`**`ldexp`**(*x*, *i*)
Return `x * (2**i)`. This is essentially the inverse of function *frexp()*.

`math.`**`modf`**(*x*)
Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

`math.`**`remainder`**(*x*, *y*)
Return the IEEE 754-style remainder of *x* with respect to *y*. For finite *x* and finite nonzero *y*, this is the difference `x - n*y`, where `n` is the closest integer to the exact value of the quotient `x / y`. If `x / y` is exactly halfway between two consecutive integers, the nearest *even* integer is used for `n`. The remainder `r = remainder(x, y)` thus always satisfies `abs(r) <= 0.5 * abs(y)`.

Special cases follow IEEE 754: in particular, `remainder(x, math.inf)` is *x* for any finite *x*, and `remainder(x, 0)` and `remainder(math.inf, x)` raise *ValueError* for any non-NaN *x*. If the result of the remainder operation is zero, that zero will have the same sign as *x*.

On platforms using IEEE 754 binary floating-point, the result of this operation is always exactly representable: no rounding error is introduced.

버전 3.7에 추가.

`math.`**`trunc`**(*x*)
Return the *Real* value *x* truncated to an *Integral* (usually an integer). Delegates to `x.__trunc__()`.

Note that *frexp()* and *modf()* have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an 'output parameter' (there is no such thing in Python).

For the *ceil()*, *floor()*, and *modf()* functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with `abs(x) >= 2**52` necessarily has no fractional bits.

9.2.2 Power and logarithmic functions

math.**exp**(*x*)
Return *e* raised to the power *x*, where *e* = 2.718281… is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

math.**expm1**(*x*)
Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision; the *expm1()* function provides a way to compute this quantity to full precision:

```
>>> 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
```

버전 3.2에 추가.

math.**log**(*x*[, *base*])
With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

math.**log1p**(*x*)
Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

math.**log2**(*x*)
Return the base-2 logarithm of *x*. This is usually more accurate than `log(x, 2)`.

버전 3.3에 추가.

더 보기:

int.bit_length() returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

math.**log10**(*x*)
Return the base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.

math.**pow**(*x*, *y*)
Return `x` raised to the power `y`. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when `x` is a zero or a NaN. If both `x` and `y` are finite, `x` is negative, and `y` is not an integer then `pow(x, y)` is undefined, and raises *ValueError*.

Unlike the built-in `**` operator, *math.pow()* converts both its arguments to type *float*. Use `**` or the built-in *pow()* function for computing exact integer powers.

math.**sqrt**(*x*)
Return the square root of *x*.

9.2.3 Trigonometric functions

`math.`**`acos`**(*x*)
Return the arc cosine of *x*, in radians.

`math.`**`asin`**(*x*)
Return the arc sine of *x*, in radians.

`math.`**`atan`**(*x*)
Return the arc tangent of *x*, in radians.

`math.`**`atan2`**(*y*, *x*)
Return `atan(y / x)`, in radians. The result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x, y)` makes this angle with the positive X axis. The point of *atan2()* is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and `atan2(1, 1)` are both `pi/4`, but `atan2(-1, -1)` is `-3*pi/4`.

`math.`**`cos`**(*x*)
Return the cosine of *x* radians.

`math.`**`hypot`**(*x*, *y*)
Return the Euclidean norm, `sqrt(x*x + y*y)`. This is the length of the vector from the origin to point `(x, y)`.

`math.`**`sin`**(*x*)
Return the sine of *x* radians.

`math.`**`tan`**(*x*)
Return the tangent of *x* radians.

9.2.4 Angular conversion

`math.`**`degrees`**(*x*)
Convert angle *x* from radians to degrees.

`math.`**`radians`**(*x*)
Convert angle *x* from degrees to radians.

9.2.5 Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

`math.`**`acosh`**(*x*)
Return the inverse hyperbolic cosine of *x*.

`math.`**`asinh`**(*x*)
Return the inverse hyperbolic sine of *x*.

`math.`**`atanh`**(*x*)
Return the inverse hyperbolic tangent of *x*.

`math.`**`cosh`**(*x*)
Return the hyperbolic cosine of *x*.

`math.`**`sinh`**(*x*)
Return the hyperbolic sine of *x*.

`math.`**`tanh`**(*x*)
Return the hyperbolic tangent of *x*.

9.2.6 Special functions

math.**erf**(*x*)
Return the error function at *x*.

The *erf()* function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

버전 3.2에 추가.

math.**erfc**(*x*)
Return the complementary error function at *x*. The complementary error function is defined as `1.0 - erf(x)`. It is used for large values of *x* where a subtraction from one would cause a loss of significance.

버전 3.2에 추가.

math.**gamma**(*x*)
Return the Gamma function at *x*.

버전 3.2에 추가.

math.**lgamma**(*x*)
Return the natural logarithm of the absolute value of the Gamma function at *x*.

버전 3.2에 추가.

9.2.7 Constants

math.**pi**
The mathematical constant $\pi = 3.141592\cdots$, to available precision.

math.**e**
The mathematical constant $e = 2.718281\cdots$, to available precision.

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!

버전 3.6에 추가.

math.**inf**
A floating-point positive infinity. (For negative infinity, use `-math.inf`.) Equivalent to the output of `float('inf')`.

버전 3.5에 추가.

math.**nan**
A floating-point "not a number" (NaN) value. Equivalent to the output of `float('nan')`.

버전 3.5에 추가.

CPython implementation detail: The *math* module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise *ValueError* for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and *OverflowError* for results that overflow (for example, `exp(1000.0)`). A NaN will not be returned from any of the functions above unless one or more of the input arguments

was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

더 보기:

Module *cmath* Complex number versions of many of these functions.

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: 부호 있는 0에 대한 하드웨어와 시스템 수준 지원이 있는 플랫폼에서, 분지 절단(branch cut)을 수반하는 함수는 분지 절단의 양 면에서 연속입니다: 0의 부호는 분지 절단의 한 면을 다른 면과 구별합니다. 부호 있는 0을 지원하지 않는 플랫폼에서 연속성은 아래에 지정된 것과 같습니다.

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. 다시 말해:

```
z == z.real + z.imag*1j
```

극좌표 *(polar coordinates)*는 복소수를 나타내는 다른 방법을 제공합니다. 극좌표에서, 복소수 z는 모듈러스(modulus) r과 위상 각(phase angle) phi로 정의됩니다. 모듈러스 r은 z에서 원점까지의 거리이며, 위상 phi는 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`**`(x)`

x의 위상(x의 편각 *(argument)*이라고도 합니다)을 float로 반환합니다. `phase(x)`는 `math.atan2(x.imag, x.real)`과 동등합니다. 결과는 $[-\pi, \pi]$ 범위에 놓이고, 이 작업의 분지 절단은 음의 실수 축에 놓이고, 위로부터 연속입니다. 부호 있는 0을 지원하는 시스템(현재 사용 중인 대부분의 시스템을 포함합니다)에서, 이는 결과의 부호가 `x.imag`가 0일 때도 `x.imag`의 부호와 같음을 의미합니다:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

참고: 복소수 x의 모듈러스(절댓값)는 내장 *abs()* 함수를 사용하여 계산할 수 있습니다. 이 연산을 위한 별도의 *cmath* 모듈 함수는 없습니다.

---

cmath.**polar**(*x*)
x 표현을 극좌표로 반환합니다. 쌍 `(r, phi)`를 반환합니다. 여기서 *r*은 *x*의 모듈러스이고 phi는 *x*의 위상입니다. `polar(x)`는 `(abs(x), phase(x))`와 동등합니다.

cmath.**rect**(*r*, *phi*)
극좌표 *r*과 *phi*를 가지는 복소수 *x*를 반환합니다. `r * (math.cos(phi) + math.sin(phi)*1j)`와 동등합니다.

9.3.2 거듭제곱과 로그 함수

cmath.**exp**(*x*)
*e*의 *x* 거듭제곱을 반환합니다. 여기서 *e*는 자연로그(natural logarithms)의 밑입니다.

cmath.**log**(*x*[, *base*])
주어진 밑*(base)*에 대한 *x*의 로그를 반환합니다. *base*가 지정되지 않으면, *x*의 자연로그를 반환합니다. 음의 실수 축을 따라 0에서부터 -∞ 까지 가고, 위로부터 연속인 하나의 분지 절단이 있습니다.

cmath.**log10**(*x*)
*x*의 밑이 10인 로그를 반환합니다. 이것은 *log()*와 같은 분지 절단을 가집니다.

cmath.**sqrt**(*x*)
*x*의 제곱근을 반환합니다. 이것은 *log()*와 같은 분지 절단을 가집니다.

9.3.3 삼각 함수

cmath.**acos**(*x*)
*x*의 아크 코사인을 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 실수 축을 따라 1에서 오른쪽으로 ∞까지 확장하고, 아래로부터 연속입니다. 다른 하나는 실수 축을 따라 -1에서 왼쪽으로 -∞까지 확장되고, 위에서부터 연속입니다.

cmath.**asin**(*x*)
*x*의 아크 사인을 반환합니다. 이것은 *acos()*와 같은 분지 절단을 가집니다.

cmath.**atan**(*x*)
*x*의 아크 탄젠트를 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 허수 축을 따라 `1j`에서 ∞`j`까지 확장되며, 오른쪽으로부터 연속입니다. 다른 하나는 허수 축을 따라 `-1j`에서 -∞`j`까지 확장되며, 왼쪽으로부터 연속입니다.

cmath.**cos**(*x*)
*x*의 코사인을 반환합니다.

cmath.**sin**(*x*)
*x*의 사인을 반환합니다.

cmath.**tan**(*x*)
*x*의 탄젠트를 반환합니다.

9.3.4 쌍곡선(hyperbolic) 함수

cmath.**acosh**(*x*)
*x*의 역 쌍곡선 코사인을 반환합니다. 하나의 분지 절단이 있습니다, 실수 축을 따라 1에서 왼쪽으로 -∞까지 확장되며 위로부터 연속입니다.

cmath.**asinh**(*x*)
*x*의 역 쌍곡선 사인을 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 허수 축을 따라 `1j`에서 `∞j`까지 확장되며, 오른쪽으로부터 연속입니다. 다른 하나는 허수 축을 따라 `-1j`에서 `-∞j`까지 확장되며, 왼쪽으로부터 연속입니다.

cmath.**atanh**(*x*)
*x*의 역 쌍곡선 탄젠트를 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 실수 축을 따라 `1`에서 `∞`까지 확장되며, 아래로부터 연속입니다. 다른 하나는 실수 축을 따라 `-1`에서 `-∞`까지 확장되며, 위로부터 연속입니다.

cmath.**cosh**(*x*)
*x*의 쌍곡선 코사인을 반환합니다.

cmath.**sinh**(*x*)
*x*의 쌍곡선 사인을 반환합니다.

cmath.**tanh**(*x*)
*x*의 쌍곡선 탄젠트를 반환합니다.

9.3.5 분류 함수

cmath.**isfinite**(*x*)
*x*의 실수부와 허수부가 모두 유한이면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

버전 3.2에 추가.

cmath.**isinf**(*x*)
*x*의 실수부나 허수부 중 하나가 무한이면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

cmath.**isnan**(*x*)
*x*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.

*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 0보다 커야 합니다.

*abs_tol*은 최소 절대 허용 오차입니다 – 0에 가까운 비교에 유용합니다. *abs_tol*은 최소한 0이어야 합니다.

에러가 발생하지 않으면, 결과는 다음과 같습니다: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

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.

버전 3.5에 추가.

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.

버전 3.6에 추가.

cmath.**inf**
부동 소수점 양의 무한대. `float('inf')`와 동등합니다.

버전 3.6에 추가.

cmath.**infj**
0 실수부와 양의 무한대 허수부를 갖는 복소수. `complex(0.0, float('inf'))`와 동등합니다.

버전 3.6에 추가.

cmath.**nan**
부동 소수점 "not a number" (NaN) 값. `float('nan')`과 동등합니다.

버전 3.6에 추가.

cmath.**nanj**
0 실수부와 NaN 허수부를 갖는 복소수. `complex(0.0, float('nan'))`과 동등합니다.

버전 3.6에 추가.

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 — 십진 고정 소수점 및 부동 소수점 산술

소스 코드: Lib/decimal.py

---

decimal 모듈은 빠르고 정확하게 자리 올림 하는 십진 부동 소수 산술을 지원합니다. *float* 데이터형보다 다음과 같은 몇 가지 장점을 제공합니다:

- 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 수는 정확하게 표현할 수 있습니다. 반면에, `1.1`과 `2.2`와 같은 수는, 이진 부동 소수점으로 정확히 표현할 수 없습니다. 최종 사용자는 일반적으로 이진 부동 소수점에서 그러하듯이 `1.1 + 2.2`가 `3.3000000000000003`처럼 표시되는 것을 기대하지 않을 것입니다.
- 정확성은 산술에서도 유지됩니다. 십진 부동 소수점에서, `0.1 + 0.1 + 0.1 - 0.3`는 정확하게 0과 같습니다. 이진 부동 소수점에서, 결과는 `5.5511151231257827e-017` 입니다. 0에 가깝지만, 차이가 신뢰할 수 있는 동등성 검사를 방해하고, 차이는 누적 될 수 있습니다. 이러한 이유로, 강한 동등성 불변 조건을 갖는 회계 응용 프로그램에서는 decimal이 선호됩니다.
- decimal 모듈은 유효 자릿수의 개념을 포함하고 있으므로 `1.30 + 1.20`은 `2.50` 입니다. 후행 0은 유효성을 나타내기 위해 유지됩니다. 이것은 화폐 응용에서는 관례적인 표현입니다. 곱셈의 경우, "교과서" 접근법은 피승수의 모든 숫자를 사용합니다. 예를 들어 `1.3 * 1.2`는 `1.56`이고, `1.30 * 1.20`은 `1.5600` 입니다.
- 하드웨어 기반 이진 부동 소수점과는 달리, decimal 모듈은 사용자가 변경할 수 있는 정밀도(기본값은 28 자리)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).

decimal 수는 불변입니다. 부호(sign), 계수(coefficient digits) 및 지수(exponent)로 구성됩니다. 유효성을 유지하기 위해, 계수는 후행 0을 자르지 않습니다. Decimal은 또한 `Infinity`, `-Infinity`, `NaN`과 같은 특별한 값을 포함합니다. 표준은 또한 `-0`을 `+0`과 구별합니다.

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:

- IBM의 일반 십진 산술 명세, The General Decimal Arithmetic Specification.

9.4.1 빠른 시작 자습서

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 인스턴스는 정수, 문자열, 실수(float) 또는 튜플로 만들 수 있습니다. 정수 나 실수로 만들면 해당 정수 또는 실수의 정확한 값 변환이 일어납니다. Decimal 수는 "숫자가 아님(Not a number)"을 나타내는 `NaN`, 양과 음의 `Infinity` 및 `-0`과 같은 특수한 값을 포함합니다:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> 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')
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
```

버전 3.3에 추가.

새로운 Decimal의 유효 숫자는 입력된 숫자의 개수에 의해서만 결정됩니다. 컨텍스트 정밀도 및 자리 올림은 오직 산술 연산 중에만 작용합니다.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

버전 3.3에서 변경.

Decimal은 파이썬의 다른 부분들과 잘 어울립니다. 다음은 십진 부동 소수점으로 부린 작은 묘기입니다:

```
>>> 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]
>>> 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('1.4142135623730950488016887242')
>>> Decimal(1).exp()
Decimal('2.7182818284590452353602874713')
>>> Decimal('10').ln()
Decimal('2.3025850929940456840179914547')
>>> Decimal('10').log10()
Decimal('1')
```

quantize() 메서드는 숫자를 고정된 지수로 자리 올림 합니다. 이 방법은 종종 결과를 고정된 자릿수로 자리 올림 하는 화폐 응용에 유용합니다.:

```
>>> 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')

>>> 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
```

컨텍스트에는 계산 중에 발생하는 예외 조건을 감시하기 위한 신호 플래그도 있습니다. 플래그는 명시적으로 지워질 때까지 설정된 상태로 유지되므로, clear_flags() 메서드를 사용하여 모니터링되는 각 계산 집합 앞에서 플래그를 지우는 것이 가장 좋습니다.

```
>>> 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=[])
```

flags 엔트리는 Pi 에 대한 유리수 근삿값이 자리 올림 되었고 (컨텍스트 정밀도 이상의 숫자가 버려졌습니다) 결과가 부정확하다는 (폐기된 숫자 일부는 0이 아닙니다) 것을 보여줍니다.

개별 트랩은 컨텍스트의 traps 필드에 있는 딕셔너리를 사용해서 설정합니다.:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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 숫자 형과 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
```

위의 `digit` 가 나타나는 곳에는 다른 유니코드 십진수도 허용됩니다. 여기에는 다양한 다른 알파벳 (예를 들어, 인도-아라비아와 데바나가리 숫자)의 십진수와 전각 숫자 `'\uff10'` 에서 `'\uff19'` 까지 포함됩니다.

value 가 *tuple* 이면, 세 개의 항목으로 구성되어야 합니다, 부호 (`0` 은 양수, `1` 은 음수), 숫자의 *tuple*, 정수 지수. 예를 들어, `Decimal((0, (1, 4, 1, 4), -3))` 은 `Decimal('1.414')` 를 반환합니다.

value 가 *float* 면, 이진 부동 소수점 값은 손실 없이 정확한 십진수로 변환됩니다. 이 변환에는 종종 53 자리 이상의 정밀도가 필요할 수 있습니다. 예를 들어, `Decimal(float('1.1'))` 은 `Decimal('1.1000000000000000888178419700125232338905334472656 25')` 로 변환됩니다.

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.

context 인자의 목적은 *value* 가 잘못된 문자열인 경우 어떻게 해야할지를 결정하는 것입니다. 컨텍스트가 *InvalidOperation* 을 트랩하면, 예외가 발생합니다; 그렇지 않으면, 생성자는 `NaN` 의 값을 갖는 새 Decimal을 반환합니다.

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.

십진 부동 소수점 객체는 *float*나 *int*와 같은 다른 내장 숫자 형과 많은 성질을 공유합니다. 일반적인 수학 연산과 특수 메서드가 모두 적용됩니다. 마찬가지로, 십진 객체는 복사, 피클, 인쇄, 딕셔너리 키로 사용, 집합 원소로 사용, 비교, 정렬 및 다른 형(가령 *float* 또는 *int*)으로 코어션될 수 있습니다.

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.

표준 숫자 속성에 더해, 십진 부동 소수점 객체에는 여러 가지 특별한 메서드가 있습니다:

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.

버전 3.6에 추가.

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 NaN 인 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')
```

from_float(*f*)
float를 십진수로 정확하게 변환하는 클래스 메서드.

Decimal.from_float(0.1) 은 *Decimal('0.1')*과 같지 않음에 유의하십시오. 0.1은 이진 부동 소수점에서 정확하게 표현할 수 없으므로, 값은 가장 가까운 표현 가능 값인 *0x1.999999999999ap-4*로 저장됩니다. 십진수로 표시된 해당 값은 *0.1000000000000000055511151231257827021181583404541015625* 입니다.

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')
```

버전 3.1에 추가.

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 or입니다.

max(*other*, *context=None*)
컨텍스트 자리 올림 규칙이 반환되기 전에 적용되고 `NaN` 값이 (컨텍스트와 신호를 주는지 조용한지에 따라) 신호를 주거나 무시되는 것을 제외하고 `max(self, other)` 와 같습니다.

max_mag(*other*, *context=None*)
max() 와 비슷하지만, 피연산자의 절댓값을 사용하여 비교가 이루어집니다.

min(*other*, *context=None*)
컨텍스트 자리 올림 규칙이 반환되기 전에 적용되고 `NaN` 값이 (컨텍스트와 신호를 주는지 조용한지에 따라) 신호를 주거나 무시되는 것을 제외하고 `min(self, other)` 와 같습니다.

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*)
가장 오른쪽 끝에 오는 0을 제거하고 결과를 `Decimal('0')`과 같은 모든 결과를 `Decimal('0e0')`으로 변환하여 숫자를 정규화합니다. 등가 클래스의 어트리뷰트에 대한 규범적인 값을 만드는 데 사용됩니다. 예를 들어, `Decimal('32.100')`과 `Decimal('0.321000e+2')`는 모두 같은 값인 `Decimal('32.1')`로 정규화됩니다.

number_class(*context=None*)
피연산자의 클래스를 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 NaN임을 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.

결과 지수가 `Emax` 보다 크거나 `Etiny` 보다 작을 때마다 에러가 반환됩니다.

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*)
self와 other가 같은 지수를 가졌는지 또는 둘 다 NaN 인지 검사합니다.

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.

논리적 피연산자

`logical_and()`, `logical_invert()`, `logical_or()` 와 `logical_xor()` 메서드는 인자가 논리적 피연산자 이길 기대합니다. 논리적 피연산자 는 지수와 부호가 모두 0이고 숫자는 모두 `0` 또는 `1` 인 *Decimal* 인스턴스입니다.

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*)
with-문으로 진입할 때 활성 스레드의 현재 컨텍스트를 *ctx* 의 복사본으로 설정하고, with-문을 빠져나올 때 이전의 컨텍스트를 복원하는 컨텍스트 관리자를 돌려줍니다. 컨텍스트를 지정하지 않으면 현재 컨텍스트의 복사본이 사용됩니다.

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
```

아래에 설명된 *Context* 생성자를 사용하여 새로운 컨텍스트를 만들 수도 있습니다. 또한, 이 모듈은 세 가지 미리 만들어진 컨텍스트를 제공합니다:

class 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.

class 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).

트랩이 비활성화되었으므로, 이 컨텍스트는 예외를 발생시키기보다 `NaN` 이나 `Infinity` 의 결괏값을 선호하는 응용 프로그램에 유용합니다. 이는 응용 프로그램이 그렇지 않으면 프로그램을 중단시킬 수 있는 조건이 있는 경우에도 실행을 완료할 수 있도록 합니다.

class 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.

기본값은 `prec=28`, `rounding=ROUND_HALF_EVEN` 이고 *Overflow*, *InvalidOperation*, *DivisionByZero* 트랩이 활성화됩니다.

3개의 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 는 컨텍스트에서 산술 연산의 정밀도를 설정하는 [1, *MAX_PREC*] 범위의 정수입니다.

rounding 옵션은 자리 올림 모드 섹션에 나열된 상수 중 하나입니다.

traps 과 *flags* 필드는 설정할 신호를 나열합니다. 일반적으로, 새 컨텍스트는 트랩만 설정하고 플래그는 지워진 채로 두어야 합니다.

Emin 과 *Emax* 필드는 지수에 허용되는 한계를 지정하는 정수입니다. *Emin* 은 [*MIN_EMIN*, 0], *Emax* 는 [0, *MAX_EMAX*] 범위 내에 있어야 합니다.

capitals 필드는 0 또는 1(기본값) 입니다. 1로 설정하면, 지수는 대문자 E와 함께 인쇄됩니다; 그렇지 않으면 소문자 e 가 사용됩니다: `Decimal('6.02e+23')`.

clamp 필드는 0 (기본값) 또는 1 입니다. 1로 설정하면, 이 컨텍스트에서 표현할 수 있는 *Decimal* 인스턴스의 지수 e 는 `Emin - prec + 1 <= e <= Emax - prec + 1` 입니다. *clamp* 가 0 이면 더 약한 조건이 유지됩니다: *Decimal* 인스턴스의 조정된 최대 Emax 입니다. *clamp* 가 1 일 때, 큰 정상 수는, 가능할 때, 지수 제약 조건을 맞추기 위해 지수가 감소하고 해당 숫자만큼의 0이 계수에 더해집니다; 이것은 수의 값을 보존하지만 유효한 후미 0에 대한 정보를 잃어버립니다. 예를 들면:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

1 의 *clamp* 값은 IEEE 754에 명시된 고정 폭 십진수 교환 형식과 호환되도록 합니다.

Context 클래스는 주어진 컨텍스트에서 직접 산술을 하는데 필요한 다수의 메서드뿐만 아니라 여러 가지 범용 메서드를 정의합니다. 이에 더해, 위에서 설명한 *Decimal* 메서드마다 (`adjusted()` 와 `as_tuple()` 메서드는 예외입니다) 대응하는 *Context* 메서드가 있습니다. 예를 들어, *Context* 인스턴스 C 와 *Decimal* 인스턴스 x 에 대해서, `C.exp(x)` 는 `x.exp(context=C)` 와 동등합니다. 각각 *Context* 메서드는 Decimal 인스턴스가 받아들여지는 곳 어디에서나 파이썬 정수(*int* 의 인스턴스)를 받아들입니다.

clear_flags()
모든 플래그를 0으로 재설정합니다.

clear_traps()
모든 트랩을 0으로 재설정합니다.

버전 3.3에 추가.

copy()
컨텍스트의 복사본을 돌려줍니다.

copy_decimal(*num*)
Decimal 인스턴스 num의 복사본을 반환합니다.

create_decimal(*num*)
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
```

버전 3.1에 추가.

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*)
e ** *x*를 반환합니다.

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 NaN이면 `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 0이면 `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.

두 인자로는 `x**y` 를 계산합니다. `x` 가 음수면 `y` 는 정수여야 합니다. `y` 가 정수이고 결과가 유한하고 'precision' 자릿수로 정확하게 표현될 수 있지 않은 이상 결과는 부정확합니다. 컨텍스트의 자리 올림 모드가 사용됩니다. 결과는 항상 파이썬 버전에서 정확하게 자리 올림 됩니다.

버전 3.3에서 변경: C 모듈은 올바르게 자리 올림 된 *exp()* 와 *ln()* 함수로 *power()* 를 계산합니다. 결과는 잘 정의되어 있지만 "거의 항상 올바르게 자리 올림 될" 뿐입니다.

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.**HAVE_THREADS**
The value is True. Deprecated, because Python now always has threads.

버전 3.9부터 폐지.

decimal.**HAVE_CONTEXTVAR**
The default value is True. If Python is compiled --without-decimal-contextvar, 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.

버전 3.9에 추가: backported to 3.7 and 3.8

9.4.5 자리 올림 모드

decimal.**ROUND_CEILING**
Infinity를 향해 올립니다.

decimal.**ROUND_DOWN**
0을 향해 자리 올림 합니다.

decimal.**ROUND_FLOOR**
-Infinity를 향해 내립니다.

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.

일반적으로, 지수가 컨텍스트의 Emin과 Emax 한계를 벗어날 때 클램핑이 발생합니다. 가능하면, 계수에 0을 추가하여 지수를 줄입니다.

class decimal.**DecimalException**
다른 신호의 베이스 클래스이고 *ArithmeticError* 의 서브 클래스입니다.

class decimal.**DivisionByZero**
무한대가 아닌 숫자를 0으로 나눴다는 신호를 줍니다.

나눗셈, 모듈로 나눗셈 또는 음수로 숫자를 거듭제곱할 때 발생할 수 있습니다. 이 신호가 트랩 되지 않으면, 계산에 제공된 입력의 부호에 따라 Infinity 나 -Infinity를 돌려줍니다.

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.

의미가 없는 연산이 요청되었음을 나타냅니다. 트랩 되지 않으면, NaN 을 반환합니다. 가능한 원인은 다음과 같습니다:

```
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.

자리 올림 후에 지수가 Emax 보다 큼을 나타냅니다. 트랩 되지 않으면, 결과는 자리 올림 모드에 따라 달라지는데, 가장 큰 표현 가능한 유한 수로 안쪽으로 당기거나 Infinity를 향해 바깥쪽으로 자리 올림 됩니다. 두 경우 모두 *Inexact*와 *Rounded* 신호도 줍니다.

class decimal.**Rounded**
정보가 손실되지는 않았지만 자리 올림이 발생했습니다.

자리 올림이 자릿수를 버릴 때마다 신호를 줍니다; 그 자릿수가 0일 때도 그렇습니다 (가령 5.00을 5.0으로 자리 올림). 트랩 되지 않으면, 결과를 그대로 반환합니다. 이 신호는 유효숫자의 손실을 감지하는 데 사용됩니다.

class decimal.**Subnormal**
자리 올림 전에 지수가 Emin 보다 작습니다.

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 부동 소수점 노트

증가시킨 정밀도로 자리 올림 오차 줄이기

십진 부동 소수점을 사용하면 십진수 표현 오차가 없어집니다 (0.1을 정확히 나타낼 수 있습니다); 그러나 0이 아닌 숫자가 고정된 정밀도를 초과할 때 일부 연산은 여전히 자리 올림 오차를 일으킬 수 있습니다.

자리 올림 오차의 효과는 거의 상쇄되는 양을 더하거나 빼는 것에 의해 증폭되어 유효숫자의 손실로 이어질 수 있습니다. Knuth는 불충분한 정밀도로 자리 올림 된 부동 소수점 산술로 인해 덧셈의 결합 법칙과 배분 법칙이 파괴되는 두 가지 사례를 제공합니다:

```
# 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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Decimal('0.01')
>>> 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

decimal 모듈의 수 체계는 NaN, sNaN, -Infinity, Infinity와 두 개의 0 +0과 -0을 포함하는 특수 값을 제공합니다.

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.

어떤 연산은 불확정적이고, NaN 을 반환하거나, *InvalidOperation* 신호가 트랩 되면, 예외를 발생시킵니다. 예를 들어, 0/0 은 "숫자가 아님 (not a number)"을 의미하는 NaN 을 반환합니다. 이 종류의 NaN 은 조용하고, 한 번 만들어지면 다른 연산에 포함될 때 항상 다른 NaN 을 생성합니다. 이 동작은 때때로 빠진 입력이 있는 일련의 계산에 유용할 수 있습니다 — 특정 결과를 잘못된 것으로 표시하면서 계산을 진행할 수 있도록 합니다.

다른 종류는 sNaN 인데, 모든 연산 후에 조용히 남아 있는 대신 신호를 줍니다. 이것은 유효하지 않은 결과가 특수한 처리를 위해 계산을 중단시켜야 할 때 유용한 반환 값입니다.

파이썬의 비교 연산자의 동작은 NaN 이 관련되어있을 때 약간 의외일 수 있습니다. 피연산자 중 하나가 조용하거나 신호를 주는 NaN 일 때, 같음 검사는 항상 *False*를 반환하고 (심지어 Decimal('NaN')==Decimal('NaN') 조차도), 다름 검사는 항상 *True*를 반환합니다. <, <=, > 또는 >= 연산자 중 하나를 사용하여 두 Decimal을 비교하려는 시도는 피연산자 중 어느 것이 든 NaN 이면 *InvalidOperation* 신호를 발생시킵니다. 이 신호가 트랩 되지 않으면 *False*를 반환합니다. 일반 십진 산술 명세는 직접 비교의 동작을 명시하지 않습니다; NaN 을 포함하는 비교를 위한 이러한 규칙은 IEEE 854 표준 (섹션 5.7의 표 3을 보세요)에서 가져온 것입니다. 엄격한 표준 준수를 위해서는, 대신 compare() 및 compare-signal() 메서드를 사용하십시오.

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.

서로 다른 부호를 갖는 부호 있는 0이 같은 것에 더해, 여전히 동등한 값이지만 다른 정밀도를 갖는 여러 표현이 존재합니다. 익숙해지는데 약간 시간이 필요합니다. 정규화된 부동 소수점 표현에 익숙한 사람들에게는, 다음 계산이 0과 같은 값을 반환한다는 것이 즉시 명백하지는 않습니다:

```
>>> 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.

setcontext() 가 *getcontext()* 전에 호출되지 않았다면, *getcontext()* 는 현재 스레드에서 사용할 새로운 컨텍스트를 자동으로 생성합니다.

새 컨텍스트는 *DefaultContext* 라는 프로토타입 컨텍스트에서 복사됩니다. 각 스레드가 응용 프로그램 전체에서 같은 값을 사용하도록 기본값을 제어하려면, *DefaultContext* 객체를 직접 수정하십시오. *getcontext()* 를 호출하는 스레드 사이에 경쟁 조건이 없도록, 어떤 스레드가 시작되기 전에 수행되어야 합니다. 예를 들면:

```
# 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)'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (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')))
    0.87758256189037271611628158260
    >>> 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')))
    0.47942553860420300027328793521
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 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
```

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. `quantize()` 메서드는 고정된 소수 자릿수로 자리 올림 합니다. *Inexact* 트랩이 설정되면, 유효성 검사에도 유용합니다:

```
>>> 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. 정수로 더하기, 빼기 및 곱하기와 같은 일부 연산은 고정 소수점을 자동으로 보존합니다. 나눗셈과 정수가 아닌 수로 곱하는 것과 같은 다른 연산은, 소수점 이하 자릿수를 바꿀 것이고, 뒤에 `quantize()` 단계를 적용할 필요가 있습니다:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

고정 소수점 응용 프로그램을 개발할 때, `quantize()` 단계를 처리하는 함수를 정의하는 것이 편리합니다:

```
>>> 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. 같은 값을 표현하는 여러 가지 방법이 있습니다. 숫자 `200`, `200.000`, `2E2`, 그리고 `02E+4` 는 모두 다양한 정밀도로 같은 값을 가집니다. 이것들은 단일하게 인식할 수 있는 표준적인 값으로 변환할 방법이 있습니까?

A. the `normalize()` 메서드는 모든 해당 값을 단일 표현으로 매핑합니다:

```
>>> 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. 일부 십진수 값은 항상 지수 표기법으로 인쇄됩니다. 지수가 아닌 표현을 얻을 방법이 있습니까?

A. 일부 값의 경우, 지수 표기법만이 계수에 있는 유효 숫자를 나타낼 수 있습니다. 예를 들어 `5.0E+3`을 `5000`으로 표현하면 값은 일정하게 유지되지만, 원본의 두 자리 유효숫자를 표시할 수 없습니다.

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. 그렇습니다. 모든 이진 부동 소수점은 Decimal로 정확히 표현될 수 있습니다. 하지만 정확한 변환이 취하는 정밀도는 직관이 제안하는 것보다 더 클 수 있습니다:

```
>>> 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. 예. CPython 과 PyPy3 구현에서, decimal 모듈의 C/CFFI 버전은 임의의 정밀도로 올바르게 자리 올림 되는 십진 부동 소수점 산술을 위한 고속 libmpdec 라이브러리를 통합합니다. libmpdec는 중간 크기의 숫자에는 카라추바 곱셈(Karatsuba multiplication)을 사용하고 매우 큰 숫자에는 수론적 변환(Number Theoretic Transform)을 사용합니다. 하지만, 이 성능 향상을 실현하려면, 컨텍스트를 자리 올림 없는 계산으로 설정 해야 합니다.

```
>>> c = getcontext()
>>> c.prec = MAX_PREC
>>> c.Emax = MAX_EMAX
>>> c.Emin = MIN_EMIN
```

버전 3.3에 추가.

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**(*other_fraction*)
class fractions.**Fraction**(*float*)
class fractions.**Fraction**(*decimal*)

class fractions.**Fraction**(*string*)

첫 번째 버전에서는 *numerator* 와 *denominator* 가 *numbers.Rational*의 인스턴스이고, numerator/denominator 값의 새 *Fraction* 인스턴스를 반환합니다. *denominator* 가 0 이면, *ZeroDivisionError*를 발생시킵니다. 두 번째 버전에서는 *other_fraction* 이 *numbers.Rational*의 인스턴스이고, 같은 값을 가진 *Fraction* 인스턴스를 반환합니다. 다음 두 버전은 *float* 나 *decimal.Decimal* 인스턴스를 받아들이고, 정확히 같은 값의 *Fraction* 인스턴스를 반환합니다. 이진 부동소수점(tut-fp-issues 참조)의 일반적인 문제로 인해, Fraction(1.1) 에 대한 인자가 정확히 11/10이 아니므로, Fraction(1.1)는 흔히 기대하듯이 Fraction(11, 10)를 반환하지 않습니다. (그러나 아래의 *limit_denominator()* 메서드에 대한 설명서를 참조하십시오.) 생성자의 마지막 버전은 문자열이나 유니코드 인스턴스를 기대합니다. 이 인스턴스의 일반적인 형식은 다음과 같습니다:

```
[sign] numerator ['/' denominator]
```

이때, 선택적 sign은 '+' 나 '-'일 수 있으며 numerator 와 denominator(있다면)는 십진수 문자열입니다. 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.

numerator

기약 분수로 나타낼 때 Fraction의 분자.

denominator

기약 분수로 나타낼 때 Fraction의 분모.

from_float(*flt*)

이 클래스 메서드는 *float* *flt*의 정확한 값을 나타내는 *Fraction*을 생성합니다. 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.

from_decimal(*dec*)
이 클래스 메서드는 *decimal.Decimal* 인스턴스 *dec*의 정확한 값을 나타내는 *Fraction*을 생성합니다.

참고: 파이썬 3.2 이상에서는, *decimal.Decimal* 인스턴스에서 직접 *Fraction* 인스턴스를 생성할 수도 있습니다.

limit_denominator(*max_denominator=1000000*)
분모가 최대 max_denominator인 self에 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* <= self를 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.

fractions.**gcd**(*a*, *b*)
정수 *a*와 *b*의 최대 공약수를 반환합니다. *a* 나 *b*가 0이 아니면, gcd(a, b)의 절댓값은 *a*와 *b*를 모두 나누는 가장 큰 정수입니다. *b*가 0이 아니면, gcd(a,b)는 *b*와 같은 부호를 가집니다; 그렇지 않으면 *a*의 부호를 취합니다. gcd(0, 0)는 0을 반환합니다.

버전 3.5부터 폐지: 대신 *math.gcd()*를 사용하십시오.

더 보기:

모듈 *numbers* 숫자 계층을 구성하는 추상 베이스 클래스.

9.6 random — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function *random()*, which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the *random.Random* class. You can instantiate your own instances of *Random* to get generators that don't share state.

Class *Random* can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the `random()`, `seed()`, `getstate()`, and `setstate()` methods. Optionally, a new generator can supply a `getrandbits()` method — this allows *randrange()* to produce selections over an arbitrarily large range.

The *random* module also provides the *SystemRandom* class which uses the system function *os.urandom()* to generate random numbers from sources provided by the operating system.

경고: The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the *secrets* module.

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.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

9.6.1 Bookkeeping functions

`random.`**`seed`**(*a=None*, *version=2*)

Initialize the random number generator.

If *a* is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the *os.urandom()* function for details on availability).

If *a* is an int, it is used directly.

With version 2 (the default), a *str*, *bytes*, or *bytearray* object gets converted to an *int* and all of its bits are used.

With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for *str* and *bytes* generates a narrower range of seeds.

버전 3.2에서 변경: Moved to the version 2 scheme which uses all of the bits in a string seed.

random.**getstate**()
Return an object capturing the current internal state of the generator. This object can be passed to *setstate()* to restore the state.

random.**setstate**(*state*)
state should have been obtained from a previous call to *getstate()*, and *setstate()* restores the internal state of the generator to what it was at the time *getstate()* was called.

random.**getrandbits**(*k*)
Returns a Python integer with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, *getrandbits()* enables *randrange()* to handle arbitrarily large ranges.

9.6.2 Functions for integers

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*[, *step*])
Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.

The positional argument pattern matches that of *range()*. Keyword arguments should not be used because the function may use them in unexpected ways.

버전 3.2에서 변경: *randrange()* is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random()*n)` which could produce slightly uneven distributions.

random.**randint**(*a*, *b*)
Return a random integer *N* such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

9.6.3 Functions for sequences

random.**choice**(*seq*)
Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises *IndexError*.

random.**choices**(*population*, *weights=None*, *, *cum_weights=None*, *k=1*)
Return a *k* sized list of elements chosen from the *population* with replacement. If the *population* is empty, raises *IndexError*.

If a *weights* sequence is specified, selections are made according to the relative weights. Alternatively, if a *cum_weights* sequence is given, the selections are made according to the cumulative weights (perhaps computed using *itertools.accumulate()*). For example, the relative weights `[10, 5, 30, 5]` are equivalent to the cumulative weights `[10, 15, 45, 50]`. Internally, the relative weights are converted to cumulative weights before making selections, so supplying the cumulative weights saves work.

If neither *weights* nor *cum_weights* are specified, selections are made with equal probability. If a weights sequence is supplied, it must be the same length as the *population* sequence. It is a *TypeError* to specify both *weights* and *cum_weights*.

The *weights* or *cum_weights* can use any numeric type that interoperates with the *float* values returned by *random()* (that includes integers, floats, and fractions but excludes decimals).

For a given seed, the *choices()* function with equal weighting typically produces a different sequence than repeated calls to *choice()*. The algorithm used by *choices()* uses floating point arithmetic for internal

consistency and speed. The algorithm used by *choice()* defaults to integer arithmetic with repeated selections to avoid small biases from round-off error.

버전 3.6에 추가.

random.**shuffle**(*x*[, *random*])
Shuffle the sequence *x* in place.

The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function *random()*.

To shuffle an immutable sequence and return a new shuffled list, use `sample(x, k=len(x))` instead.

Note that even for small `len(x)`, the total number of permutations of *x* can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

random.**sample**(*population*, *k*)
Return a *k* length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be *hashable* or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample from a range of integers, use a *range()* object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), k=60)`.

If the sample size is larger than the population size, a *ValueError* is raised.

9.6.4 Real-valued distributions

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

random.**random**()
Return the next random floating point number in the range [0.0, 1.0).

random.**uniform**(*a*, *b*)
Return a random floating point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the equation `a + (b-a) * random()`.

random.**triangular**(*low*, *high*, *mode*)
Return a random floating point number *N* such that `low <= N <= high` and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

random.**betavariate**(*alpha*, *beta*)
Beta distribution. Conditions on the parameters are `alpha > 0` and `beta > 0`. Returned values range between 0 and 1.

random.**expovariate**(*lambd*)
Exponential distribution. *lambd* is 1.0 divided by the desired mean. It should be nonzero. (The parameter would

be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity if *lambd* is positive, and from negative infinity to 0 if *lambd* is negative.

`random.`**`gammavariate`**(*alpha*, *beta*)
Gamma distribution. (*Not* the gamma function!) Conditions on the parameters are `alpha > 0` and `beta > 0`.

The probability distribution function is:

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

`random.`**`gauss`**(*mu*, *sigma*)
Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the *normalvariate()* function defined below.

`random.`**`lognormvariate`**(*mu*, *sigma*)
Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

`random.`**`normalvariate`**(*mu*, *sigma*)
Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

`random.`**`vonmisesvariate`**(*mu*, *kappa*)
mu is the mean angle, expressed in radians between 0 and 2**pi*, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2**pi*.

`random.`**`paretovariate`**(*alpha*)
Pareto distribution. *alpha* is the shape parameter.

`random.`**`weibullvariate`**(*alpha*, *beta*)
Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

9.6.5 Alternative Generator

`class` `random.`**`Random`**([*seed*])
Class that implements the default pseudo-random number generator used by the *random* module.

`class` `random.`**`SystemRandom`**([*seed*])
Class that uses the *os.urandom()* function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are not reproducible. Accordingly, the *seed()* method has no effect and is ignored. The *getstate()* and *setstate()* methods raise *NotImplementedError* if called.

9.6.6 Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo random number generator. By re-using a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module's algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator's `random()` method will continue to produce the same sequence when the compatible seeder is given the same seed.

9.6.7 Examples and Recipes

Basic examples:

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5␣
↪seconds
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)   # Four samples without replacement
[40, 10, 50, 30]
```

Simulations:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck of 52 playing cards
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> def trial():
...     return choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
...
>>> sum(trial() for i in range(10000)) / 10000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2500 <= sorted(choices(range(10000), k=5))[2] < 7500
...
>>> sum(trial() for i in range(10000)) / 10000
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
0.7958
```

Example of statistical bootstrapping using resampling with replacement to estimate a confidence interval for the mean of a sample of size five:

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in range(20))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[1]:.1f} to {means[-2]:.1f}')
```

Example of a resampling permutation test to determine the statistical significance or p-value of an observed difference between the effects of a drug versus a placebo:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import 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 = 10000
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.')
```

Simulation of arrival times and service deliveries in a single server queue:

```
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 5.0
stdev_service_time = 0.5

num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else arrival
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)

waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.')
```

더 보기:

Statistics for Hackers a video tutorial by Jake Vanderplas on statistical analysis using just a few fundamental concepts including simulation, sampling, shuffling, and cross-validation.

Economics Simulation a simulation of a marketplace by Peter Norvig that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) a tutorial by Peter Norvig covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.

9.7 `statistics` — Mathematical statistics functions

버전 3.4에 추가.

Source code: Lib/statistics.py

This module provides functions for calculating mathematical statistics of numeric (`Real`-valued) data.

참고: Unless explicitly noted otherwise, these functions support *int*, *float*, *decimal.Decimal* and *fractions.Fraction*. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Mixed types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use *map()* to ensure a consistent result, e.g. `map(float, input_data)`.

9.7.1 Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

mean()	Arithmetic mean ("average") of data.
harmonic_mean()	Harmonic mean of data.
median()	Median (middle value) of data.
median_low()	Low median of data.
median_high()	High median of data.
median_grouped()	Median, or 50th percentile, of grouped data.
mode()	Mode (most common value) of discrete data.

9.7.2 Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

pstdev()	Population standard deviation of data.
pvariance()	Population variance of data.
stdev()	Sample standard deviation of data.
variance()	Sample variance of data.

9.7.3 Function details

Note: The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

`statistics.`**`mean`**(*data*)

Return the sample arithmetic mean of *data* which can be a sequence or iterator.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called "the average", although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, *StatisticsError* will be raised.

Some examples of use:

```
>>> 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: The mean is strongly affected by outliers and is not a robust estimator for central location: the mean is not necessarily a typical example of the data points. For more robust, although less efficient, measures of central location, see *median()* and *mode()*. (In this case, "efficient" refers to statistical efficiency rather than computational efficiency.)

The sample mean gives an unbiased estimate of the true population mean, which means that, taken on average over all the possible samples, `mean(sample)` converges on the true mean of the entire population. If *data* represents the entire population rather than a sample, then `mean(data)` is equivalent to calculating the true population mean μ.

`statistics.`**`harmonic_mean`**(*data*)

Return the harmonic mean of *data*, a sequence or iterator of real-valued numbers.

The harmonic mean, sometimes called the subcontrary mean, is the reciprocal of the arithmetic *mean()* of the reciprocals of the data. For example, the harmonic mean of three values *a*, *b* and *c* will be equivalent to `3/(1/a + 1/b + 1/c)`.

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging quantities which are rates or ratios, for example speeds. For example:

Suppose an investor purchases an equal value of shares in each of three companies, with P/E (price/earning) ratios of 2.5, 3 and 10. What is the average P/E ratio for the investor's portfolio?

```
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment portfolio.
3.6
```

Using the arithmetic mean would give an average of about 5.167, which is too high.

StatisticsError is raised if *data* is empty, or any element is less than zero.

버전 3.6에 추가.

statistics.**median**(*data*)

Return the median (middle value) of numeric data, using the common "mean of middle two" method. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

The median is a robust measure of central location, and is less affected by the presence of outliers in your data. When the number of data points is odd, the middle data point is returned:

```
>>> median([1, 3, 5])
3
```

When the number of data points is even, the median is interpolated by taking the average of the two middle values:

```
>>> median([1, 3, 5, 7])
4.0
```

This is suited for when your data is discrete, and you don't mind that the median may not be an actual data point.

If your data is ordinal (supports order operations) but not numeric (doesn't support addition), you should use *median_low()* or *median_high()* instead.

더 보기:

median_low(), *median_high()*, *median_grouped()*

statistics.**median_low**(*data*)

Return the low median of numeric data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

The low median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the smaller of the two middle values is returned.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use the low median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.**median_high**(*data*)

Return the high median of data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

The high median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the larger of the two middle values is returned.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use the high median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.**median_grouped**(*data*, *interval=1*)

Return the median of grouped continuous data, calculated as the 50th percentile, using interpolation. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

In the following example, the data are rounded, so that each value represents the midpoint of data classes, e.g. 1 is the midpoint of the class 0.5–1.5, 2 is the midpoint of 1.5–2.5, 3 is the midpoint of 2.5–3.5, etc. With the data given, the middle value falls somewhere in the class 3.5–4.5, and interpolation is used to estimate it:

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

Optional argument *interval* represents the class interval, and defaults to 1. Changing the class interval naturally will change the interpolation:

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

This function does not check whether the data points are at least *interval* apart.

CPython implementation detail: Under some circumstances, *median_grouped()* may coerce data points to floats. This behaviour is likely to change in the future.

더 보기:

- "Statistics for the Behavioral Sciences", Frederick J Gravetter and Larry B Wallnau (8th Edition).
- The SSMEDIAN function in the Gnome Gnumeric spreadsheet, including this discussion.

statistics.**mode**(*data*)

Return the most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value, and is a robust measure of central location.

If *data* is empty, or if there is not exactly one most common value, *StatisticsError* is raised.

`mode` assumes discrete data, and returns a single value. This is the standard treatment of the mode as commonly taught in schools:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic which also applies to nominal (non-numeric) data:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

statistics.**pstdev**(*data*, *mu=None*)

Return the population standard deviation (the square root of the population variance). See *pvariance()* for arguments and other details.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

Return the population variance of *data*, a non-empty iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *mu* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function to calculate the variance from the entire population. To estimate the variance from a sample, the *variance()* function is usually a better choice.

Raises *StatisticsError* if *data* is empty.

Examples:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *mu* to avoid recalculation:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

This function does not attempt to verify that you have passed the actual mean as *mu*. Using arbitrary values for *mu* may lead to invalid or impossible results.

Decimals and Fractions are supported:

```
>>> 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: When called with the entire population, this gives the population variance σ². When called on a sample instead, this is the biased sample variance s², also known as variance with N degrees of freedom.

If you somehow know the true population mean μ, you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are representative (e.g. independent and identically distributed), the result will be an unbiased estimate of the population variance.

---

statistics.**stdev**(*data*, *xbar=None*)

Return the sample standard deviation (the square root of the sample variance). See *variance()* for arguments and other details.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

Return the sample variance of *data*, an iterable of at least two real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *xbar* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function when your data is a sample from a population. To calculate the variance from the entire population, see *pvariance()*.

Raises *StatisticsError* if *data* has fewer than two values.

Examples:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *xbar* to avoid recalculation:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

This function does not attempt to verify that you have passed the actual mean as *xbar*. Using arbitrary values for *xbar* can lead to invalid or impossible results.

Decimal and Fraction values are supported:

```
>>> 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: This is the sample variance s² with Bessel's correction, also known as variance with N-1 degrees of freedom. Provided that the data points are representative (e.g. independent and identically distributed), the result should be an unbiased estimate of the true population variance.

If you somehow know the actual population mean μ you should pass it to the *pvariance()* function as the *mu* parameter to get the variance of a sample.

9.7.4 Exceptions

A single exception is defined:

exception statistics.**StatisticsError**
Subclass of *ValueError* for statistics-related exceptions.

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

This module implements a number of *iterator* building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining *`map()`* and *`count()`* to form `map(f, count())`.

These tools and their built-in counterparts also work well with the high-speed functions in the *`operator`* module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: `sum(map(operator.mul, vector1, vector2))`.

Infinite iterators:

Iterator	Arguments	Results	Example
`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, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ⋯	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
chain()	p, q, ⋯	p0, p1, ⋯ plast, q0, q1, ⋯	`chain('ABC', 'DEF') --> A B C D E F`
chain.from_iterable()	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()	pred, seq	seq[n], seq[n+1], starting when pred fails	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
filterfalse()	pred, seq	elements of seq where pred(elem) is false	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
groupby()	iterable[, key]	sub-iterators grouped by value of key(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ⋯	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
takewhile()	pred, seq	seq[0], seq[1], until pred fails	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
tee()	it, n	it1, it2, ⋯ itn splits one iterator into n	
zip_longest()	p, q, ⋯	(p[0], q[0]), (p[1], q[1]), ⋯	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

Combinatoric iterators:

Iterator	Arguments	Results
product()	p, q, ⋯ [repeat=1]	cartesian product, equivalent to a nested for-loop
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements
combinations()	p, r	r-length tuples, in sorted order, no repeated elements
combinations_with_replacement()	p, r	r-length tuples, in sorted order, with repeated elements
`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 module 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*[, *func*])

Make an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional *func* argument). If *func* is supplied, it should be a function of two arguments. Elements of the input *iterable* may be any type that can be accepted as arguments to *func*. (For example, with the default operation of addition, elements may be any addable type including `Decimal` or `Fraction`.) If the input iterable is empty, the output iterable will also be empty.

Roughly equivalent to:

```
def accumulate(iterable, func=operator.add):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

There are a number of uses for the *func* argument. It can be set to `min()` for a running minimum, `max()` for a running maximum, or `operator.mul()` for a running product. Amortization tables can be built by accumulating interest and applying payments. First-order recurrence relations can be modeled by supplying the initial value in the iterable and using only the accumulated total in *func* argument:

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _:  r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)     # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

See `functools.reduce()` for a similar function that returns only the final accumulated value.

버전 3.2에 추가.

버전 3.3에서 변경: Added the optional *func* parameter.

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. Used for treating consecutive sequences as a single sequence. Roughly equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.**from_iterable**(*iterable*)

Alternate constructor for *chain()*. Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

itertools.**combinations**(*iterable*, *r*)

Return *r* length subsequences of elements from the input *iterable*.

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

Roughly equivalent to:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    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)
```

The code for *combinations()* can be also expressed as a subsequence of *permutations()* after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```
def combinations(iterable, r):
    pool = tuple(iterable)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

The number of items returned is `n! / r! / (n-r)!` when `0 <= r <= n` or zero when `r > n`.

itertools.**combinations_with_replacement**(*iterable*, *r*)

Return *r* length subsequences of elements from the input *iterable* allowing individual elements to be repeated more than once.

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, the generated combinations will also be unique.

Roughly equivalent to:

```
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)
```

The code for *combinations_with_replacement()* can be also expressed as a subsequence of *product()* after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

The number of items returned is `(n+r-1)! / r! / (n-1)!` when `n > 0`.

버전 3.1에 추가.

itertools.**compress**(*data*, *selectors*)

Make an iterator that filters elements from *data* returning only those that have a corresponding element in *selectors* that evaluates to `True`. Stops when either the *data* or *selectors* iterables has been exhausted. Roughly equivalent to:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

버전 3.1에 추가.

itertools.**count**(*start=0*, *step=1*)

Make an iterator that returns evenly spaced values starting with number *start*. Often used as an argument to *map()* to generate consecutive data points. Also, used 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: Added *step* argument and allowed non-integer arguments.

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
```

Note, this member of the toolkit 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 as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce *any* output until the predicate first becomes false, so it may have a lengthy start-up time. Roughly equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.**filterfalse**(*predicate*, *iterable*)

Make an iterator that filters elements from iterable returning only those for which the predicate is `False`. If *predicate* is `None`, return the items that are false. Roughly equivalent to:

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

Make an iterator that returns consecutive keys and groups from the *iterable*. The *key* is a function computing a key value for each element. If not specified or is `None`, *key* defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of *groupby()* is similar to the `uniq` filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with *groupby()*. Because the source is shared, when the *groupby()* object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```
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() is roughly equivalent to:

```
class groupby:
    # [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
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
        self.id = object()
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey, self.id))
    def _grouper(self, tgtkey, id):
        while self.id is id and self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.**islice**(*iterable*, *stop*)
itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

Make an iterator that returns selected elements from the iterable. If *start* is non-zero, then elements from the iterable are skipped until start is reached. Afterward, elements are returned consecutively unless *step* is set higher than one which results in items being skipped. If *stop* is None, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position. Unlike regular slicing, *islice()* does not support negative values for *start*, *stop*, or *step*. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line). Roughly equivalent to:

```
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, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

If *start* is None, then iteration starts at zero. If *step* is None, then the step defaults to one.

itertools.**permutations**(*iterable*, *r=None*)

Return successive *r* length permutations of elements in the *iterable*.

If *r* is not specified or is None, then *r* defaults to the length of the *iterable* and all possible full-length permutations are generated.

Permutations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the permutation tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each permutation.

Roughly equivalent to:

```
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))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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
```

The code for *permutations()* can be also expressed as a subsequence of *product()*, filtered to exclude entries with repeated elements (those from the same position in the input pool):

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

The number of items returned is `n! / (n-r)!` when `0 <= r <= n` or zero when `r > n`.

itertools.**product**(**iterables*, *repeat=1*)

Cartesian product of input iterables.

Roughly equivalent to nested for-loops in a generator expression. For example, `product(A, B)` returns the same as `((x,y) for x in A for y in B)`.

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional *repeat* keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`.

This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```
def product(*args, 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
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

itertools.**repeat**(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified.

Used as argument to *map()* for invariant parameters to the called function. Also used with *zip()* to create an invariant part of a tuple record.

Roughly equivalent to:

```
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 are already grouped in tuples from a single iterable (the data has been "pre-zipped"). 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, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.**tee**(*iterable*, *n=2*)

Return *n* independent iterators from a single iterable.

The following Python code helps explain what *tee* does (although the actual implementation is more complex and uses only a single underlying FIFO queue).

Roughly equivalent to:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                try:
                    newval = next(it)   # fetch a new value and
                except StopIteration:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                    return
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Once *tee()* has made a split, the original *iterable* should not be used anywhere else; otherwise, the *iterable* could get advanced without the tee objects being informed.

`tee` iterators are not threadsafe. A *RuntimeError* may be raised when using simultaneously iterators returned by the same *tee()* call, even if the original *iterable* is threadsafe.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use *list()* instead of *tee()*.

`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*. Iteration continues until the longest iterable is exhausted. Roughly equivalent to:

```
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            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()*). If not specified, *fillvalue* defaults to `None`.

10.1.2 Itertools Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The extended tools offer the same high performance as the underlying toolset. The 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 which helps eliminate temporary variables. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and *generator*s which incur interpreter overhead.

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.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:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        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 all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    'Use a predicate to partition entries into false entries and true entries'
    # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "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 unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        iter_except(functools.partial(heappop, h), IndexError)    # priority queue␣
↪iterator
        iter_except(d.popitem, KeyError)                          # non-blocking dict␣
↪iterator
        iter_except(d.popleft, IndexError)                        # non-blocking deque␣
↪iterator
        iter_except(q.get_nowait, Queue.Empty)                    # loop over a␣
↪producer Queue
        iter_except(s.pop, KeyError)                              # non-blocking set␣
↪iterator

    """
    try:
        if first is not None:
            yield first()            # For database APIs needing an initial cast to␣
↪db.first()
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # 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(pred, iterable), default)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    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):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)
```

Note, many of the above recipes can be optimized by replacing global lookups with local variables defined as default values. For example, the *dotproduct* recipe can be written as:

```
def dotproduct(vec1, vec2, sum=sum, map=map, mul=operator.mul):
    return sum(map(mul, vec1, vec2))
```

10.2 functools — Higher-order functions and operations on callable objects

Source code: Lib/functools.py

The *functools* module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The *functools* module defines the following functions:

functools.**cmp_to_key**(*func*)

Transform an old-style comparison function to a *key function*. Used with tools that accept key functions (such as *sorted()*, *min()*, *max()*, *heapq.nlargest()*, *heapq.nsmallest()*, *itertools.groupby()*). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.

A comparison function is any callable that accept two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

Example:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

For sorting examples and a brief sorting tutorial, see sortinghowto.

버전 3.2에 추가.

@functools.**lru_cache**(*maxsize=128*, *typed=False*)

Decorator to wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be hashable.

Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, *f(a=1, b=2)* and *f(b=2, a=1)* differ in their keyword argument order and may have two separate cache entries.

If *maxsize* is set to None, the LRU feature is disabled and the cache can grow without bound. The LRU feature performs best when *maxsize* is a power-of-two.

If *typed* is set to true, function arguments of different types will be cached separately. For example, f(3) and f(3.0) will be treated as distinct calls with distinct results.

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*. In a multi-threaded environment, the hits and misses are approximate.

The decorator also provides a cache_clear() function for clearing or invalidating the cache.

The original underlying function is accessible through the __wrapped__ attribute. This is useful for introspection, for bypassing the cache, or for rewrapping the function with a different cache.

An LRU (least recently used) cache works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache's size limit assures that the cache does not grow without bound on long-running processes such as web servers.

In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call, or impure functions such as time() or random().

Example of an LRU cache for static web content:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    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)
```

Example of efficiently computing Fibonacci numbers using a cache to implement a dynamic programming technique:

```
@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)
```

버전 3.2에 추가.

버전 3.3에서 변경: Added the *typed* option.

`@functools.`**`total_ordering`**

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

The class must define one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. In addition, the class should supply an `__eq__()` method.

For example:

```
@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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

참고: While this decorator makes it easy to create well behaved totally ordered types, it *does* come at the cost of slower execution and more complex stack traces for the derived comparison methods. If performance benchmarking indicates this is a bottleneck for a given application, implementing all six rich comparison methods instead is likely to provide an easy speed boost.

버전 3.2에 추가.

버전 3.4에서 변경: Returning NotImplemented from the underlying comparison function for unrecognised types is now supported.

`functools.`**`partial`**(*func*, **args*, ***keywords*)

Return a new *partial object* which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The *partial()* 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 two:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

class `functools.`**`partialmethod`**(*func*, **args*, ***keywords*)

Return a new *partialmethod* descriptor which behaves like *partial* except that it is designed to be used as a method definition rather than being directly callable.

func must be a *descriptor* or a callable (objects which are both, like normal functions, are handled as descriptors).

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.

When *func* is a non-descriptor callable, an appropriate bound method is created dynamically. This behaves like a normal Python function when used as a method: the *self* argument will be inserted as the first positional argument, even before the *args* and *keywords* supplied to the *partialmethod* constructor.

Example:

```
>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         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
```

버전 3.4에 추가.

`functools.`**`reduce`**(*function*, *iterable*[, *initializer*])

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `(((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *sequence*. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initializer* is not given and *sequence* contains only one item, the first item is returned.

Roughly equivalent to:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

`@functools.`**`singledispatch`**

Transform a function into a *single-dispatch generic function*.

To define a generic function, decorate it with the `@singledispatch` decorator. Note that the dispatch happens on the type of the first argument, create your function accordingly:

```
>>> 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. It is 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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     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)
```

For code which doesn't use type annotations, the appropriate type argument can be passed explicitly to the decorator itself:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

To enable registering lambdas and pre-existing functions, the `register()` attribute can 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 which enables decorator stacking, pickling, as well as creating 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
```

When called, the generic function dispatches on the type of the first argument:

```
>>> 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with `@singledispatch` is registered for the base `object` type, which means it is used if no better implementation is found.

To check which implementation will the generic function choose for a given type, use the `dispatch()` attribute:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

To access all registered implementations, use the read-only `registry` attribute:

```
>>> 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>
```

버전 3.4에 추가.

버전 3.7에서 변경: The `register()` attribute supports using type annotations.

`functools.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Update a *wrapper* function to look like the *wrapped* function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants `WRAPPER_ASSIGNMENTS` (which assigns to the wrapper function's `__module__`, `__name__`, `__qualname__`, `__annotations__` and `__doc__`, the documentation string) and `WRAPPER_UPDATES` (which updates the wrapper function's `__dict__`, i.e. the instance dictionary).

To allow access to the original function for introspection and other purposes (e.g. bypassing a caching decorator such as *`lru_cache()`*), this function automatically adds a `__wrapped__` attribute to the wrapper that refers to the function being wrapped.

The main intended use for this function is in *decorator* functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

`update_wrapper()` may be used with callables other than functions. Any attributes named in *assigned* or *updated* that are missing from the object being wrapped are ignored (i.e. this function will not attempt to set them on the wrapper function). *`AttributeError`* is still raised if the wrapper function itself is missing any attributes named in *updated*.

버전 3.2에 추가: Automatic addition of the `__wrapped__` attribute.

버전 3.2에 추가: Copying of the `__annotations__` attribute by default.

버전 3.2에서 변경: Missing attributes no longer trigger an *`AttributeError`*.

버전 3.4에서 변경: The __wrapped__ attribute now always refers to the wrapped function, even if that function defined a __wrapped__ attribute. (see bpo-17482)

@functools.**wraps**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

This is a convenience function for invoking *update_wrapper()* as a function decorator when defining a wrapper function. It is equivalent to partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated). For example:

```
>>> 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
>>> 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 Objects

partial objects are callable objects created by *partial()*. They have three read-only attributes:

partial.**func**

A callable object or function. Calls to the *partial* object will be forwarded to *func* with new arguments and keywords.

partial.**args**

The leftmost positional arguments that will be prepended to the positional arguments provided to a *partial* object call.

partial.**keywords**

The keyword arguments that will be supplied when the *partial* object is called.

partial objects are like function objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the *__name__* and __doc__ attributes are not created automatically. Also, *partial* objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.

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.

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.

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.

버전 3.5에 추가.

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 - b를 반환합니다.

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
a / b를 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 + b를 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*)
o 객체의 추정된 길이를 반환합니다. 먼저 실제 길이를 반환하려고 시도한 다음, object.__length_hint__()를 사용하여 추정치를 반환하려고 하고, 마지막으로 default 값을 반환합니다.

버전 3.4에 추가.

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.name을 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

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))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*[, *args...*])

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)`
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))`

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

연산	문법	함수
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 <<= b와 동등합니다.

operator.**imod**(*a*, *b*)
operator.**__imod__**(*a*, *b*)
a = imod(a, b)는 a %= b와 동등합니다.

operator.**imul**(*a*, *b*)
operator.**__imul__**(*a*, *b*)
a = imul(a, b)는 a *= b와 동등합니다.

operator.**imatmul**(*a*, *b*)
operator.**__imatmul__**(*a*, *b*)
a = imatmul(a, b)는 a @= b와 동등합니다.

버전 3.5에 추가.

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

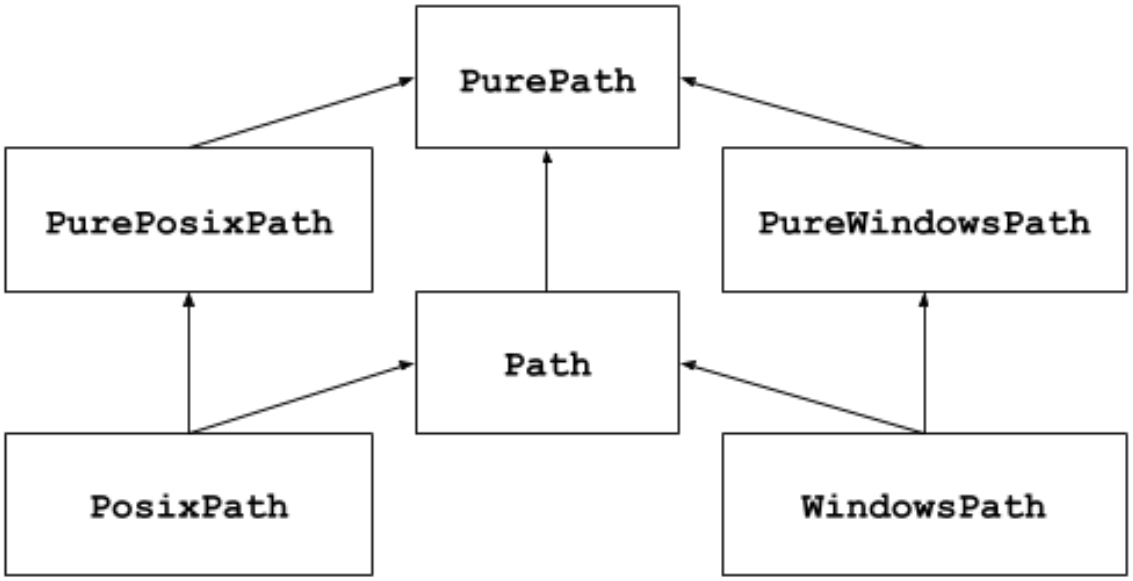
이 장에서 설명하는 모듈은 디스크 파일과 디렉터리를 다룹니다. 예를 들어, 파일의 속성을 읽고, 이식성 있는 방식으로 경로를 조작하고, 임시 파일을 만드는 모듈이 있습니다. 이 장의 전체 모듈 목록은 다음과 같습니다:

11.1 `pathlib` — Object-oriented filesystem paths

버전 3.4에 추가.

Source code: Lib/pathlib.py

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between *pure paths*, which provide purely computational operations without I/O, and *concrete paths*, which inherit from pure paths but also provide I/O operations.



If you've never used this module before or just aren't sure which class is right for your task, `Path` is most likely what you need. It instantiates a *concrete path* for the platform the code is running on.

Pure paths are useful in some special cases; for example:

1. If you want to manipulate Windows paths on a Unix machine (or vice versa). You cannot instantiate a `WindowsPath` when running on Unix, but you can instantiate `PureWindowsPath`.
2. You want to make sure that your code only manipulates paths without actually accessing the OS. In this case, instantiating one of the pure classes may be useful since those simply don't have any OS-accessing operations.

더 보기:

PEP 428: The pathlib module – object-oriented filesystem paths.

더 보기:

For low-level path manipulation on strings, you can also use the `os.path` module.

11.1.1 Basic use

Importing the main class:

```
>>> from pathlib import Path
```

Listing subdirectories:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Listing Python source files in this directory tree:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Navigating inside a directory tree:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Querying path properties:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Opening a file:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Pure paths

Pure path objects provide path-handling operations which don't actually access a filesystem. There are three ways to access these classes, which we also call *flavours*:

class pathlib.**PurePath**(**pathsegments*)

A generic class that represents the system's path flavour (instantiating it creates either a *PurePosixPath* or a *PureWindowsPath*):

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of *pathsegments* can be either a string representing a path segment, an object implementing the *os.PathLike* interface which returns a string, or another path object:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

When *pathsegments* is empty, the current directory is assumed:

```
>>> PurePath()
PurePosixPath('.')
```

When several absolute paths are given, the last is taken as an anchor (mimicking *os.path.join()*'s behaviour):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

However, in a Windows path, changing the local root doesn't discard the previous drive setting:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Spurious slashes and single dots are collapsed, but double dots (`'..'`) are not, since this would change the meaning of a path in the face of symbolic links:

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(a naïve approach would make `PurePosixPath('foo/../bar')` equivalent to `PurePosixPath('bar')`, which is wrong if `foo` is a symbolic link to another directory)

Pure path objects implement the *os.PathLike* interface, allowing them to be used anywhere the interface is accepted.

버전 3.6에서 변경: Added support for the *os.PathLike* interface.

class pathlib.**PurePosixPath**(**pathsegments*)

A subclass of *PurePath*, this path flavour represents non-Windows filesystem paths:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**PureWindowsPath**(**pathsegments*)

A subclass of *PurePath*, this path flavour represents Windows filesystem paths:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

pathsegments is specified similarly to *PurePath*.

Regardless of the system you're running on, you can instantiate all of these classes, since they don't provide any operation that does system calls.

General properties

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')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Paths of a different flavour compare unequal and cannot be ordered:

```
>>> 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
↪'
```

Operators

The slash operator helps create child paths, similarly to *os.path.join()*:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

A path object can be used anywhere an object implementing *os.PathLike* is accepted:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

The string representation of a path is the raw filesystem path itself (in native form, e.g. with backslashes under Windows), which you can pass to any function taking a file path as a string:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

Similarly, calling *bytes* on a path gives the raw filesystem path as a bytes object, as encoded by *os.fsencode()*:

```
>>> bytes(p)
b'/etc'
```

참고: Calling *bytes* is only recommended under Unix. Under Windows, the unicode form is the canonical representation of filesystem paths.

Accessing individual parts

To access the individual "parts" (components) of a path, use the following property:

`PurePath.`**`parts`**
A tuple giving access to the path's various components:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(note how the drive and local root are regrouped in a single part)

Methods and properties

Pure paths provide the following methods and properties:

`PurePath.`**`drive`**
A string representing the drive letter or name, if any:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC shares are also considered drives:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

`PurePath.`**`root`**
A string representing the (local or global) root, if any:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC shares always have a root:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

`PurePath.`**`anchor`**
The concatenation of the drive and root:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**

An immutable sequence providing access to the logical ancestors of the path:

```
>>> 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:/')
```

PurePath.**parent**

The logical parent of the path:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

You cannot go past an anchor, or empty path:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

참고: This is a purely lexical operation, hence the following behaviour:

```
>>> 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**

A string representing the final path component, excluding the drive and root, if any:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC drive names are not considered:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**
The file extension of the final component, if any:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.**suffixes**
A list of the path's file extensions:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.**stem**
The final path component, without its suffix:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()
Return a string representation of the path with forward slashes (/):

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**as_uri**()
Represent the path as a `file` URI. *ValueError* is raised if the path isn't absolute.

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.**is_absolute**()
Return whether the path is absolute or not. A path is considered absolute if it has both a root and (if the flavour allows) a drive:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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_reserved**()

With *PureWindowsPath*, return `True` if the path is considered reserved under Windows, `False` otherwise. With *PurePosixPath*, `False` is always returned.

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

File system calls on reserved paths can fail mysteriously or have unintended effects.

PurePath.**joinpath**(**other*)

Calling this method is equivalent to combining the path with each of the *other* arguments 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.**match**(*pattern*)

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise.

If *pattern* is relative, the path can be either relative or absolute, and matching is done from the right:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

If *pattern* is absolute, the path must be absolute, and the whole path must match:

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

As with other methods, case-sensitivity follows platform defaults:

```
>>> PurePosixPath('b.py').match('*.PY')
False
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> PureWindowsPath('b.py').match('*.PY')
True
```

PurePath.**relative_to**(**other*)

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 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/usr'
```

PurePath.**with_name**(*name*)

Return a new path with the *name* changed. If the original path doesn't have a name, ValueError is raised:

```
>>> 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')
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_suffix**(*suffix*)

Return a new path with the *suffix* changed. If the original path doesn't have a suffix, the new *suffix* is appended instead. If the *suffix* is an empty string, the original suffix is removed:

```
>>> 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')
```

11.1.3 Concrete paths

Concrete paths are subclasses of the pure path classes. In addition to operations provided by the latter, they also provide methods to do system calls on path objects. There are three ways to instantiate concrete paths:

class pathlib.**Path**(**pathsegments*)
A subclass of *PurePath*, this class represents concrete paths of the system's path flavour (instantiating it creates either a *PosixPath* or a *WindowsPath*):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**PosixPath**(**pathsegments*)
A subclass of *Path* and *PurePosixPath*, this class represents concrete non-Windows filesystem paths:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**WindowsPath**(**pathsegments*)
A subclass of *Path* and *PureWindowsPath*, this class represents concrete Windows filesystem paths:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments is specified similarly to *PurePath*.

You can only instantiate the class flavour that corresponds to your system (allowing system calls on non-compatible path flavours could lead to bugs or failures in your application):

```
>>> 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__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

Methods

Concrete paths provide the following methods in addition to pure paths methods. Many of these methods can raise an *OSError* if a system call fails (for example because the path doesn't exist):

classmethod Path.**cwd**()
Return a new path object representing the current directory (as returned by *os.getcwd()*):

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod Path.**home**()
Return a new path object representing the user's home directory (as returned by *os.path.expanduser()* with ~ construct):

```
>>> Path.home()
PosixPath('/home/antoine')
```

버전 3.5에 추가.

Path.**stat**()
Return a *os.stat_result* object containing information about this path, like *os.stat()*. The result is looked up at each call to this method.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Path.**chmod**(*mode*)
Change the file mode and permissions, like *os.chmod()*:

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Path.**exists**()
Whether the path points to an existing file or directory:

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

참고: If the path points to a symlink, *exists()* returns whether the symlink *points to* an existing file or directory.

Path.**expanduser**()
Return a new path with expanded ~ and ~user constructs, as returned by *os.path.expanduser()*:

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

버전 3.5에 추가.

Path.**glob**(*pattern*)
Glob the given relative *pattern* in the directory represented by this path, yielding all matching files (of any kind):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

The "`**`" pattern means "this directory and all subdirectories, recursively". In other words, it enables recursive globbing:

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

참고: Using the "`**`" pattern in large directory trees may consume an inordinate amount of time.

`Path.`**`group`**`()`
Return the name of the group owning the file. *KeyError* is raised if the file's gid isn't found in the system database.

`Path.`**`is_dir`**`()`
Return `True` if the path points to a directory (or a symbolic link pointing to a directory), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.`**`is_file`**`()`
Return `True` if the path points to a regular file (or a symbolic link pointing to a regular file), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`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. Not implemented on Windows.

버전 3.7에 추가.

`Path.`**`is_symlink`**`()`
Return `True` if the path points to a symbolic link, `False` otherwise.

`False` is also returned if the path doesn't exist; other errors (such as permission errors) are propagated.

`Path.`**`is_socket`**`()`
Return `True` if the path points to a Unix socket (or a symbolic link pointing to a Unix socket), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.`**`is_fifo`**`()`
Return `True` if the path points to a FIFO (or a symbolic link pointing to a FIFO), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.`**`is_block_device`**`()`
Return `True` if the path points to a block device (or a symbolic link pointing to a block device), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.`**`is_char_device`**`()`
Return `True` if the path points to a character device (or a symbolic link pointing to a character device), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.`**`iterdir`**`()`
When the path points to a directory, yield path objects of the directory contents:

```
>>> 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')
```

`Path.`**`lchmod`**`(`*mode*`)`
Like *`Path.chmod()`* but, if the path points to a symbolic link, the symbolic link's mode is changed rather than its target's.

`Path.`**`lstat`**`()`
Like *`Path.stat()`* but, if the path points to a symbolic link, return the symbolic link's information rather than its target's.

`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' `umask` value to determine the file mode and access flags. If the path already exists, *`FileExistsError`* is raised.

If *parents* is true, any missing parents of this path are created as needed; they are created with the default permissions without taking *mode* into account (mimicking the POSIX `mkdir -p` command).

If *parents* is false (the default), a missing parent raises *`FileNotFoundError`*.

If *exist_ok* is false (the default), *`FileExistsError`* is raised if the target directory already exists.

If *exist_ok* is true, *`FileExistsError`* exceptions will be ignored (same behavior as the POSIX `mkdir -p` command), but only if the last path component is not an existing non-directory file.

버전 3.5에서 변경: The *exist_ok* parameter was added.

`Path.`**`open`**`(`*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*`)`
Open the file pointed to by the path, like the built-in *`open()`* function does:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
'#!/usr/bin/env python3\n'
```

Path.**owner**()
Return the name of the user owning the file. *KeyError* is raised if the file's uid isn't found in the system database.

Path.**read_bytes**()
Return the binary contents of the pointed-to file as a bytes object:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

버전 3.5에 추가.

Path.**read_text**(*encoding=None*, *errors=None*)
Return the decoded contents of the pointed-to file as a string:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

The file is opened and then closed. The optional parameters have the same meaning as in *open()*.

버전 3.5에 추가.

Path.**rename**(*target*)
Rename this file or directory to the given *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. *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)
>>> target.open().read()
'some text'
```

Path.**replace**(*target*)
Rename this file or directory to the given *target*. If *target* points to an existing file or directory, it will be unconditionally replaced.

Path.**resolve**(*strict=False*)
Make the path absolute, resolving any symlinks. A new path object is returned:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

"`..`" components are also eliminated (this is the only method to do so):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

If the path doesn't exist and *strict* is `True`, *FileNotFoundError* is raised. If *strict* is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists. If an infinite loop is encountered along the resolution path, *RuntimeError* is raised.

버전 3.6에 추가: The *strict* argument (pre-3.6 behavior is strict).

`Path.`**`rglob`**(*pattern*)

This is like calling *Path.glob()* with "`**/`" added in front of the given relative *pattern*:

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

`Path.`**`rmdir`**()

Remove this directory. The directory must be empty.

`Path.`**`samefile`**(*other_path*)

Return whether this path points to the same file as *other_path*, which can be either a Path object, or a string. The semantics are similar to *os.path.samefile()* and *os.path.samestat()*.

An *OSError* can be raised if either file cannot be accessed for some reason.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

버전 3.5에 추가.

`Path.`**`symlink_to`**(*target*, *target_is_directory=False*)

Make this path a symbolic link to *target*. Under Windows, *target_is_directory* must be true (default `False`) if the link's target is a directory. Under POSIX, *target_is_directory*'s value 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: The order of arguments (link, target) is the reverse of *os.symlink()*'s.

`Path.`**`touch`**(*mode=0o666*, *exist_ok=True*)

Create a file at this given path. If *mode* is given, it is combined with the process' `umask` value to determine the file mode and access flags. If the file already exists, the function succeeds if *exist_ok* is true (and its modification time is updated to the current time), otherwise *FileExistsError* is raised.

Path.**unlink**()
Remove this file or symbolic link. If the path points to a directory, use *Path.rmdir()* instead.

Path.**write_bytes**(*data*)
Open the file pointed to in bytes mode, write *data* to it, and close the file:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

An existing file of the same name is overwritten.

버전 3.5에 추가.

Path.**write_text**(*data*, *encoding=None*, *errors=None*)
Open the file pointed to in text mode, write *data* to it, and close the file:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

버전 3.5에 추가.

11.1.4 Correspondence to tools in the os module

Below is a table mapping various *os* functions to their corresponding *PurePath*/*Path* equivalent.

참고: Although *os.path.relpath()* and *PurePath.relative_to()* have some overlapping use-cases, their semantics differ enough to warrant not considering them equivalent.

os and os.path	pathlib
os.path.abspath()	*Path.resolve()*
os.chmod()	*Path.chmod()*
os.mkdir()	*Path.mkdir()*
os.rename()	*Path.rename()*
os.replace()	*Path.replace()*
os.rmdir()	*Path.rmdir()*
os.remove(), *os.unlink()*	*Path.unlink()*
os.getcwd()	*Path.cwd()*
os.path.exists()	*Path.exists()*
os.path.expanduser()	*Path.expanduser()* and *Path.home()*
os.path.isdir()	*Path.is_dir()*
os.path.isfile()	*Path.is_file()*
os.path.islink()	*Path.is_symlink()*
os.stat()	*Path.stat()*, *Path.owner()*, *Path.group()*
os.path.isabs()	*PurePath.is_absolute()*
os.path.join()	*PurePath.joinpath()*
os.path.basename()	*PurePath.name*
os.path.dirname()	*PurePath.parent*
os.path.samefile()	*Path.samefile()*
os.path.splitext()	*PurePath.suffix*

11.2 `os.path` — Common pathname manipulations

Source code: Lib/posixpath.py (for POSIX), Lib/ntpath.py (for Windows NT), and Lib/macpath.py (for Macintosh)

This module implements some useful functions on pathnames. To read or write files see *open()*, and for accessing the filesystem see the *os* module. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings. Unfortunately, some file names may not be representable as strings on Unix, so applications that need to support arbitrary file names on Unix should use bytes objects to represent path names. Vice versa, using bytes objects cannot represent all file names on Windows (in the standard `mbcs` encoding), hence Windows applications should use string objects to access all files.

Unlike a unix shell, Python does not do any *automatic* path expansions. Functions such as *expanduser()* and *expandvars()* can be invoked explicitly when an application desires shell-like path expansion. (See also the *glob* module.)

더 보기:

The *pathlib* module offers high-level path objects.

참고: All of these functions accept either only bytes or only string objects as their parameters. The result is an object of the same type, if a path or file name is returned.

참고: Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The *os.path* module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is *always* in one of the different formats. They all have the same interface:

- `posixpath` for UNIX-style paths
- `ntpath` for Windows paths
- *`macpath`* for old-style MacOS paths

`os.path.`**`abspath`**(*path*)
Return a normalized absolutized version of the pathname *path*. On most platforms, this is equivalent to calling the function *`normpath()`* as follows: `normpath(join(os.getcwd(), path))`.

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`basename`**(*path*)
Return the base name of pathname *path*. This is the second element of the pair returned by passing *path* to the function *`split()`*. Note that the result of this function is different from the Unix **`basename`** program; where **`basename`** for `'/foo/bar/'` returns `'bar'`, the *`basename()`* function returns an empty string (`''`).

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`commonpath`**(*paths*)
Return the longest common sub-path of each pathname in the sequence *paths*. Raise *`ValueError`* if *paths* contains both absolute and relative pathnames, or if *paths* is empty. Unlike *`commonprefix()`*, this returns a valid path.

Availability: Unix, Windows.

버전 3.5에 추가.

버전 3.6에서 변경: Accepts a sequence of *path-like objects*.

`os.path.`**`commonprefix`**(*list*)
Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string (`''`).

참고: This function may return invalid paths because it works a character at a time. To obtain a valid path, see *`commonpath()`*.

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`dirname`**(*path*)
Return the directory name of pathname *path*. This is the first element of the pair returned by passing *path* to the function *`split()`*.

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`exists`**(*path*)
Return `True` if *path* refers to an existing path or an open file descriptor. Returns `False` for broken symbolic links. On some platforms, this function may return `False` if permission is not granted to execute *`os.stat()`* on the requested file, even if the *path* physically exists.

버전 3.3에서 변경: *path* can now be an integer: `True` is returned if it is an open file descriptor, `False` otherwise.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**lexists**(*path*)
Return True if *path* refers to an existing path. Returns True for broken symbolic links. Equivalent to *exists()* on platforms lacking *os.lstat()*.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**expanduser**(*path*)
On Unix and Windows, return the argument with an initial component of ~ or ~user replaced by that *user*'s home directory.

On Unix, an initial ~ is replaced by the environment variable HOME if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module *pwd*. An initial ~user is looked up directly in the password directory.

On Windows, HOME and USERPROFILE will be used if set, otherwise a combination of HOMEPATH and HOMEDRIVE will be used. An initial ~user is handled by stripping the last directory component from the created user path derived above.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**expandvars**(*path*)
Return the argument with environment variables expanded. Substrings of the form $name or ${name} are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, %name% expansions are supported in addition to $name and ${name}.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**getatime**(*path*)
Return the time of last access of *path*. The return value is a floating point number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

os.path.**getmtime**(*path*)
Return the time of last modification of *path*. The return value is a floating point number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**getctime**(*path*)
Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**getsize**(*path*)
Return the size, in bytes, of *path*. Raise *OSError* if the file does not exist or is inaccessible.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**isabs**(*path*)
Return True if *path* is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with a (back)slash after chopping off a potential drive letter.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**isfile**(*path*)
Return True if *path* is an *existing* regular file. This follows symbolic links, so both *islink()* and *isfile()* can be true for the same path.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**isdir**(*path*)
Return True if *path* is an *existing* directory. This follows symbolic links, so both *islink()* and *isdir()* can be true for the same path.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**islink**(*path*)
Return True if *path* refers to an *existing* directory entry that is a symbolic link. Always False if symbolic links are not supported by the Python runtime.

버전 3.6에서 변경: Accepts a *path-like object*.

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 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 3.4에 추가: Support for detecting non-root mount points on Windows.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**join**(*path*, **paths*)
Join one or more path components intelligently. The return value is the concatenation of *path* and any members of **paths* with exactly one directory separator (os.sep) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.

On Windows, the drive letter is not reset when an absolute path component (e.g., r'\foo') is encountered. If a component contains a drive letter, all previous components are thrown away and the drive letter is reset. Note that since there is a current directory for each drive, os.path.join("c:", "foo") represents a path relative to the current directory on drive C: (c:foo), not c:\foo.

버전 3.6에서 변경: Accepts a *path-like object* for *path* and *paths*.

os.path.**normcase**(*path*)
Normalize the case of a pathname. On Windows, convert all characters in the pathname to lowercase, and also convert forward slashes to backward slashes. On other operating systems, return the path unchanged. Raise a *TypeError* if the type of *path* is not str or bytes (directly or indirectly through the *os.PathLike* interface).

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**normpath**(*path*)
Normalize a pathname by collapsing redundant separators and up-level references so that A//B, A/B/, A/./B and A/foo/../B all become A/B. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use *normcase()*.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**realpath**(*path*)
Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system).

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**relpath**(*path*, *start=os.curdir*)
Return a relative filepath to *path* either from the current directory or from an optional *start* directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of *path* or *start*.

start defaults to `os.curdir`.

Availability: Unix, Windows.

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`samefile`**(*path1*, *path2*)
Return `True` if both pathname arguments refer to the same file or directory. This is determined by the device number and i-node number and raises an exception if an `os.stat()` call on either pathname fails.

Availability: Unix, Windows.

버전 3.2에서 변경: Added Windows support.

버전 3.4에서 변경: Windows now uses the same implementation as all other platforms.

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`sameopenfile`**(*fp1*, *fp2*)
Return `True` if the file descriptors *fp1* and *fp2* refer to the same file.

Availability: Unix, Windows.

버전 3.2에서 변경: Added Windows support.

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`samestat`**(*stat1*, *stat2*)
Return `True` if the stat tuples *stat1* and *stat2* refer to the same file. These structures may have been returned by `os.fstat()`, `os.lstat()`, or `os.stat()`. This function implements the underlying comparison used by `samefile()` and `sameopenfile()`.

Availability: Unix, Windows.

버전 3.4에서 변경: Added Windows support.

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`split`**(*path*)
Split the pathname *path* into a pair, `(head, tail)` where *tail* is the last pathname component and *head* is everything leading up to that. The *tail* part will never contain a slash; if *path* ends in a slash, *tail* will be empty. If there is no slash in *path*, *head* will be empty. If *path* is empty, both *head* and *tail* are empty. Trailing slashes are stripped from *head* unless it is the root (one or more slashes only). In all cases, `join(head, tail)` returns a path to the same location as *path* (but the strings may differ). Also see the functions `dirname()` and `basename()`.

버전 3.6에서 변경: Accepts a *path-like object*.

`os.path.`**`splitdrive`**(*path*)
Split the pathname *path* into a pair `(drive, tail)` where *drive* is either a mount point or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, `drive + tail` will be the same as *path*.

On Windows, splits a pathname into drive/UNC sharepoint and relative path.

If the path contains a drive letter, drive will contain everything up to and including the colon. e.g. `splitdrive("c:/dir")` returns `("c:", "/dir")`

If the path contains a UNC path, drive will contain the host name and share, up to but not including the fourth separator. e.g. `splitdrive("//host/computer/dir")` returns `("//host/computer", "/dir")`

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**splitext**(*path*)
Split the pathname *path* into a pair `(root, ext)` such that `root + ext == path`, and *ext* is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; `splitext('.cshrc')` returns `('.cshrc', '')`.

버전 3.6에서 변경: Accepts a *path-like object*.

os.path.**supports_unicode_filenames**
`True` if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

11.3 `fileinput` — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see *open()*.

The typical use is:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is `'-'`, it is also replaced by `sys.stdin` and the optional arguments *mode* and *openhook* are ignored. To specify an alternative list of filenames, pass it as the first argument to *input()*. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to *input()* or *FileInput*. If an I/O error occurs during opening or reading a file, *OSError* is raised.

버전 3.3에서 변경: *IOError* used to be raised; it is now an alias of *OSError*.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the *openhook* parameter to *fileinput.input()* or *FileInput()*. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module:

fileinput.**input**(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)
Create an instance of the *FileInput* class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the *FileInput* class.

The *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

버전 3.2에서 변경: Can be used as a context manager.

Deprecated since version 3.6, will be removed in version 3.8: The *bufsize* parameter.

The following functions use the global state created by `fileinput.input()`; if there is no active state, `RuntimeError` is raised.

`fileinput.filename()`
Return the name of the file currently being read. Before the first line has been read, returns `None`.

`fileinput.fileno()`
Return the integer "file descriptor" for the current file. When no file is opened (before the first line and between files), returns `-1`.

`fileinput.lineno()`
Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

`fileinput.filelineno()`
Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

`fileinput.isfirstline()`
Return `True` if the line just read is the first line of its file, otherwise return `False`.

`fileinput.isstdin()`
Return `True` if the last line was read from `sys.stdin`, otherwise return `False`.

`fileinput.nextfile()`
Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

`fileinput.close()`
Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

class `fileinput.FileInput`(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)
Class `FileInput` is the implementation; its methods `filename()`, `fileno()`, `lineno()`, `filelineno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it has a `readline()` method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With *mode* you can specify which file mode will be passed to `open()`. It must be one of `'r'`, `'rU'`, `'U'` and `'rb'`.

The *openhook*, when given, must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. You cannot use *inplace* and *openhook* together.

A `FileInput` instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

버전 3.2에서 변경: Can be used as a context manager.

버전 3.4부터 폐지: The `'rU'` and `'U'` modes.

Deprecated since version 3.6, will be removed in version 3.8: The *bufsize* parameter.

Optional in-place filtering: if the keyword argument `inplace=True` is passed to *fileinput.input()* or to the *FileInput* constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module:

`fileinput.`**`hook_compressed`**(*filename*, *mode*)
Transparently opens files compressed with gzip and bzip2 (recognized by the extensions `'.gz'` and `'.bz2'`) using the *gzip* and *bz2* modules. If the filename extension is not `'.gz'` or `'.bz2'`, the file is opened normally (ie, using *open()* without any decompression).

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

`fileinput.`**`hook_encoded`**(*encoding*, *errors=None*)
Returns a hook which opens each file with *open()*, using the given *encoding* and *errors* to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

버전 3.6에서 변경: Added the optional *errors* parameter.

11.4 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.

버전 3.4에 추가.

stat.**S_ISPORT**(*mode*)
mode가 이벤트 포트(event port)로부터 왔으면 0이 아닌 값을 반환합니다.

버전 3.4에 추가.

stat.**S_ISWHT**(*mode*)
mode가 화이트 아웃(whiteout)으로부터 왔으면 0이 아닌 값을 반환합니다.

버전 3.4에 추가.

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):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            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.

버전 3.3에 추가.

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.

버전 3.4에 추가.

stat.**S_IFPORT**
이벤트 포트.

버전 3.4에 추가.

stat.**S_IFWHT**
화이트 아웃(whiteout).

버전 3.4에 추가.

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_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 X 10.6+).

stat.**UF_HIDDEN**
파일을 GUI에 표시하면 안 됩니다 (맥 OS X 10.5+).

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_NOUNLINK**
파일의 이름을 변경하거나 삭제할 수 없습니다.

stat.**SF_SNAPSHOT**
파일은 스냅숏(snapshot) 파일입니다.

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**
버전 3.5에 추가.

11.5 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 True를 반환하고, 그렇지 않으면 False를 반환합니다.

*shallow*가 참이면 같은 *os.stat()* 서명을 갖는 파일을 같다고 취급합니다. 그렇지 않으면 파일의 내용을 비교합니다.

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/c와 b/c, a/d/e와 b/d/e를 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.

버전 3.4에 추가.

11.5.1 dircmp 클래스

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*)
*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 *filecmp.cmp()*에서 설명한 대로 얕은*(shallow)* 비교를 수행하여 파일을 비교합니다.

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* 객체로 매핑하는 딕셔너리.

filecmp.**DEFAULT_IGNORES**
버전 3.4에 추가.

*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)
```

11.6 tempfile — Generate temporary files and directories

Source code: Lib/tempfile.py

This module creates temporary files and directories. It works on all supported platforms. *TemporaryFile*, *NamedTemporaryFile*, *TemporaryDirectory*, and *SpooledTemporaryFile* are high-level interfaces which provide automatic cleanup and can be used as context managers. *mkstemp()* and *mkdtemp()* are lower-level functions which require manual cleanup.

All the user-callable functions and constructors take additional arguments which allow direct control over the location and name of temporary files and directories. Files names used by this module include a string of random characters which allows those files to be securely created in shared temporary directories. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable items:

tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)
Return a *file-like object* that can be used as a temporary storage area. The file is created securely, using the same rules as *mkstemp()*. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is either not created at all or is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The resulting object can be used as a context manager (see *Examples*). On completion of the context or destruction of the file object the temporary file will be removed from the filesystem.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *buffering*, *encoding* and *newline* are interpreted as for `open()`.

The *dir*, *prefix* and *suffix* parameters have the same meaning and defaults as with `mkstemp()`.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose `file` attribute is the underlying true file object.

The `os.O_TMPFILE` flag is used if it is available and works (Linux-specific, requires Linux kernel 3.11 or later).

버전 3.5에서 변경: The `os.O_TMPFILE` flag is now used if available.

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*)

This function operates exactly as `TemporaryFile()` does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the `name` attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If *delete* is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file.

tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)

This function operates exactly as `TemporaryFile()` does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with `TemporaryFile()`.

The resulting file has one additional method, `rollover()`, which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose `_file` attribute is either an `io.BytesIO` or `io.TextIOWrapper` object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

버전 3.3에서 변경: the truncate method now accepts a `size` argument.

tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*)

This function securely creates a temporary directory using the same rules as `mkdtemp()`. The resulting object can be used as a context manager (see *Examples*). On completion of the context or destruction of the temporary directory object the newly created temporary directory and all its contents are removed from the filesystem.

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a context manager, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there is one.

The directory can be explicitly cleaned up by calling the `cleanup()` method.

버전 3.2에 추가.

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the `os.O_EXCL` flag for `os.open()`. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike `TemporaryFile()`, the user of `mkstemp()` is responsible for deleting the temporary file when done with it.

If *suffix* is not None, the file name will end with that suffix, otherwise there will be no suffix. *mkstemp()* does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is not None, the file name will begin with that prefix; otherwise, a default prefix is used. The default is the return value of *gettempprefix()* or *gettempprefixb()*, as appropriate.

If *dir* is not None, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the *TMPDIR*, *TEMP* or *TMP* environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via os.popen().

If any of *suffix*, *prefix*, and *dir* are not None, they must be the same type. If they are bytes, the returned name will be bytes instead of str. If you want to force a bytes return value with otherwise default behavior, pass suffix=b''.

If *text* is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by *os.open()*) and the absolute pathname of that file, in that order.

버전 3.5에서 변경: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to None to cause an appropriate default value to be used.

버전 3.6에서 변경: The *dir* parameter now accepts a *path-like object*.

tempfile.**mkdtemp**(*suffix=None*, *prefix=None*, *dir=None*)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of *mkdtemp()* is responsible for deleting the temporary directory and its contents when done with it.

The *prefix*, *suffix*, and *dir* arguments are the same as for *mkstemp()*.

mkdtemp() returns the absolute pathname of the new directory.

버전 3.5에서 변경: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to None to cause an appropriate default value to be used.

버전 3.6에서 변경: The *dir* parameter now accepts a *path-like object*.

tempfile.**gettempdir**()

Return the name of the directory used for temporary files. This defines the default value for the *dir* argument to all functions in this module.

Python searches a standard list of directories to find one which the calling user can create files in. The list is:

1. The directory named by the TMPDIR environment variable.
2. The directory named by the TEMP environment variable.
3. The directory named by the TMP environment variable.
4. A platform-specific location:
   - On Windows, the directories C:\TEMP, C:\TMP, \TEMP, and \TMP, in that order.
   - On all other platforms, the directories /tmp, /var/tmp, and /usr/tmp, in that order.
5. As a last resort, the current working directory.

The result of this search is cached, see the description of *tempdir* below.

tempfile.**gettempdirb**()
Same as *gettempdir()* but the return value is in bytes.

버전 3.5에 추가.

tempfile.**gettempprefix**()
Return the filename prefix used to create temporary files. This does not contain the directory component.

tempfile.**gettempprefixb**()
Same as *gettempprefix()* but the return value is in bytes.

버전 3.5에 추가.

The module uses a global variable to store the name of the directory used for temporary files returned by *gettempdir()*. It can be set directly to override the selection process, but this is discouraged. All functions in this module take a *dir* argument which can be used to specify the directory and this is the recommended approach.

tempfile.**tempdir**
When set to a value other than `None`, this variable defines the default value for the *dir* argument to the functions defined in this module.

If `tempdir` is `None` (the default) at any call to any of the above functions except *gettempprefix()* it is initialized following the algorithm described in *gettempdir()*.

11.6.1 Examples

Here are some examples of typical usage of the *tempfile* module:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 Deprecated functions and variables

A historical way to create temporary files was to first generate a file name with the *mktemp()* function and then create a file using this name. Unfortunately this is not secure, because a different process may create a file with this name in the time between the call to *mktemp()* and the subsequent attempt to create the file by the first process. The solution is to combine the two steps and create the file immediately. This approach is used by *mkstemp()* and the other functions described above.

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

버전 2.3부터 폐지: Use *mkstemp()* instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are similar to those of *mkstemp()*, except that bytes file names, `suffix=None` and `prefix=None` are not supported.

경고: Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. *mktemp()* usage can be replaced easily with *NamedTemporaryFile()*, passing it the `delete=False` parameter:

```
>>> 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
```

11.7 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()*와 달리, *glob*은 점(`.`)으로 시작하는 파일 이름을 특수한 경우로 취급합니다. (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.

더 보기:

pathlib 모듈은 고수준의 경로 객체를 제공합니다.

glob.**glob**(*pathname*, ***, *recursive=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 포함됩니다(셸과 마찬가지로).

If *recursive* is true, the pattern "`**`" will match any files and zero or more directories, subdirectories and symbolic links to directories. If the pattern is followed by an *os.sep* or *os.altsep* then files will not match.

> **참고:** 커다란 디렉터리 트리에서 "**" 패턴을 사용하면 과도한 시간이 걸릴 수 있습니다.

버전 3.5에서 변경: "**" 를 사용하는 재귀적 glob 지원.

glob.**iglob**(*pathname*, *, *recursive=False*)
실제로 동시에 저장하지 않고 *glob()*과 같은 값을 산출하는 이터레이터를 반환합니다.

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.

버전 3.4에 추가.

예를 들어, 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:

모듈 ***fnmatch*** 셸 스타일 파일명 (경로가 아님) 확장

11.8 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.

fnmatch.**fnmatch**(*filename*, *pattern*)

filename 문자열이 *pattern* 문자열과 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**(*filename*, *pattern*)

*filename*이 *pattern*과 일치하는지를 검사하여, *True* 나 *False*를 반환합니다; 비교는 대소 문자를 구분하며, *os.path.normcase()*를 적용하지 않습니다.

fnmatch.**filter**(*names*, *pattern*)

*pattern*에 일치하는 *names* 리스트의 부분 집합을 반환합니다. [n for n in names if fnmatch(n, pattern)]과 같지만, 더 효율적으로 구현됩니다.

fnmatch.**translate**(*pattern*)

셸 스타일의 *pattern*을 *re.match()*에서 사용하기 위해 정규식으로 변환한 값을 반환합니다.

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.

11.9 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).

  *filename*이라는 파일이 없으면, 이 함수는 먼저 *module_globals*에서 **PEP 302** `__loader__`를 먼저 확인한 후(zip 파일이나 기타 파일 시스템 이외의 임포트 소스에서 모듈이 임포트 됩니다) 모듈 검색 경로, `sys.path`, 에서 파일을 찾습니다.

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.

  버전 3.5에 추가.

예제:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 shutil — High-level file operations

Source code: Lib/shutil.py

The *shutil* module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the *os* module.

경고: Even the higher-level file copying functions (`shutil.copy()`, `shutil.copy2()`) cannot copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

11.10.1 Directory and files operations

`shutil.`**`copyfileobj`**(*fsrc*, *fdst*[, *length*])

Copy the contents of the file-like object *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

`shutil.`**`copyfile`**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst*. *src* and *dst* are path names given as strings. *dst* must be the complete target file name; look at `shutil.copy()` for a copy that accepts a target directory path. If *src* and *dst* specify the same file, `SameFileError` is raised.

The destination location must be writable; otherwise, an `OSError` exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If *follow_symlinks* is false and *src* is a symbolic link, a new symbolic link will be created instead of copying the file *src* points to.

버전 3.3에서 변경: `IOError` used to be raised instead of `OSError`. Added *follow_symlinks* argument. Now returns *dst*.

버전 3.4에서 변경: Raise `SameFileError` instead of `Error`. Since the former is a subclass of the latter, this change is backward compatible.

`exception` `shutil.`**`SameFileError`**

This exception is raised if source and destination in `copyfile()` are the same file.

버전 3.4에 추가.

`shutil.`**`copymode`**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings. If *follow_symlinks* is false, and both *src* and *dst* are symbolic links, `copymode()` will attempt to modify the mode of *dst* itself (rather than the file it points to). This functionality is not available on every platform; please see `copystat()` for more information. If `copymode()` cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

버전 3.3에서 변경: Added *follow_symlinks* argument.

`shutil.`**`copystat`**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. On Linux, `copystat()` also copies the "extended attributes" where possible. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings.

If *follow_symlinks* is false, and *src* and *dst* both refer to symbolic links, `copystat()` will operate on the symbolic links themselves rather than the files the symbolic links refer to—reading the information from the *src* symbolic link, and writing the information to the *dst* symbolic link.

참고: Not all platforms provide the ability to examine and modify symbolic links. Python itself can tell you what functionality is locally available.

- If `os.chmod in os.supports_follow_symlinks` is `True`, *copystat()* can modify the permission bits of a symbolic link.
- If `os.utime in os.supports_follow_symlinks` is `True`, *copystat()* can modify the last access and modification times of a symbolic link.
- If `os.chflags in os.supports_follow_symlinks` is `True`, *copystat()* can modify the flags of a symbolic link. (`os.chflags` is not available on all platforms.)

On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, *copystat()* will copy everything it can. *copystat()* never returns failure.

Please see *os.supports_follow_symlinks* for more information.

버전 3.3에서 변경: Added *follow_symlinks* argument and support for Linux extended attributes.

shutil.**copy**(*src*, *dst*, *, *follow_symlinks=True*)

Copies the file *src* to the file or directory *dst*. *src* and *dst* should be strings. If *dst* specifies a directory, the file will be copied into *dst* using the base filename from *src*. Returns the path to the newly created file.

If *follow_symlinks* is false, and *src* is a symbolic link, *dst* will be created as a symbolic link. If *follow_symlinks* is true and *src* is a symbolic link, *dst* will be a copy of the file *src* refers to.

copy() copies the file data and the file's permission mode (see *os.chmod()*). Other metadata, like the file's creation and modification times, is not preserved. To preserve all file metadata from the original, use *copy2()* instead.

버전 3.3에서 변경: Added *follow_symlinks* argument. Now returns path to the newly created file.

shutil.**copy2**(*src*, *dst*, *, *follow_symlinks=True*)

Identical to *copy()* except that *copy2()* also attempts to preserve file metadata.

When *follow_symlinks* is false, and *src* is a symbolic link, *copy2()* attempts to copy all metadata from the *src* symbolic link to the newly-created *dst* symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, *copy2()* will preserve all the metadata it can; *copy2()* never returns failure.

copy2() uses *copystat()* to copy the file metadata. Please see *copystat()* for more information about platform support for modifying symbolic link metadata.

버전 3.3에서 변경: Added *follow_symlinks* argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

shutil.**ignore_patterns**(**patterns*)

This factory function creates a function that can be used as a callable for *copytree()*'s *ignore* argument, ignoring files and directories that match one of the glob-style *patterns* provided. See the example below.

shutil.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*)

Recursively copy an entire directory tree rooted at *src*, returning the destination directory. The destination directory, named by *dst*, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with *copystat()*, individual files are copied using *shutil.copy2()*.

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When *symlinks* is false, if the file pointed by the symlink doesn't exist, an exception will be added in the list of errors raised in an *Error* exception at the end of the copy process. You can set the optional *ignore_dangling_symlinks* flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don't support *os.symlink()*.

If *ignore* is given, it must be a callable that will receive as its arguments the directory being visited by *copytree()*, and a list of its contents, as returned by *os.listdir()*. Since *copytree()* is called recursively, the *ignore* callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. *ignore_patterns()* can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an *Error* is raised with a list of reasons.

If *copy_function* is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, *shutil.copy2()* is used, but any function that supports the same signature (like *shutil.copy()*) can be used.

버전 3.3에서 변경: Copy metadata when *symlinks* is false. Now returns *dst*.

버전 3.2에서 변경: Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silent dangling symlinks errors when *symlinks* is false.

shutil.**rmtree**(*path*, *ignore_errors=False*, *onerror=None*)

Delete an entire directory tree; *path* must point to a directory (but not a symbolic link to a directory). If *ignore_errors* is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by *onerror* or, if that is omitted, they raise an exception.

참고: On platforms that support the necessary fd-based functions a symlink attack resistant version of *rmtree()* is used by default. On other platforms, the *rmtree()* implementation is susceptible to a symlink attack: given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn't be able to access otherwise. Applications can use the *rmtree.avoids_symlink_attacks* function attribute to determine which case applies.

If *onerror* is provided, it must be a callable that accepts three parameters: *function*, *path*, and *excinfo*.

The first parameter, *function*, is the function which raised the exception; it depends on the platform and implementation. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, will be the exception information returned by *sys.exc_info()*. Exceptions raised by *onerror* will not be caught.

버전 3.3에서 변경: Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

rmtree.**avoids_symlink_attacks**

Indicates whether the current platform and implementation provides a symlink attack resistant version of *rmtree()*. Currently this is only true for platforms supporting fd-based directory access functions.

버전 3.3에 추가.

shutil.**move**(*src*, *dst*, *copy_function=copy2*)

Recursively move a file or directory (*src*) to another location (*dst*) and return the destination.

If the destination is an existing directory, then *src* is moved inside that directory. If the destination already exists but is not a directory, it may be overwritten depending on *os.rename()* semantics.

If the destination is on the current filesystem, then *os.rename()* is used. Otherwise, *src* is copied to *dst* using *copy_function* and then removed. In case of symlinks, a new symlink pointing to the target of *src* will be created in or as *dst* and *src* will be removed.

If *copy_function* is given, it must be a callable that takes two arguments *src* and *dst*, and will be used to copy *src* to *dest* if *os.rename()* cannot be used. If the source is a directory, *copytree()* is called, passing it the `copy_function()`. The default *copy_function* is *copy2()*. Using *copy()* as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

버전 3.3에서 변경: Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU's **mv**. Now returns *dst*.

버전 3.5에서 변경: Added the *copy_function* keyword argument.

`shutil.`**`disk_usage`**(*path*)
Return disk usage statistics about the given path as a *named tuple* with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. On Windows, *path* must be a directory; on Unix, it can be a file or directory.

버전 3.3에 추가.

Availability: Unix, Windows.

`shutil.`**`chown`**(*path*, *user=None*, *group=None*)
Change owner *user* and/or *group* of the given *path*.

user can be a system user name or a uid; the same applies to *group*. At least one argument is required.

See also `os.chown()`, the underlying function.

Availability: Unix.

버전 3.3에 추가.

`shutil.`**`which`**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)
Return the path to an executable which would be run if the given *cmd* was called. If no *cmd* would be called, return `None`.

mode is a permission mask passed to `os.access()`, by default determining if the file exists and executable.

When no *path* is specified, the results of `os.environ()` are used, returning either the "PATH" value or a fallback of `os.defpath`.

On Windows, the current directory is always prepended to the *path* whether or not you use the default or provide your own, which is the behavior the command shell uses when finding executables. Additionally, when finding the *cmd* in the *path*, the `PATHEXT` environment variable is checked. For example, if you call `shutil.which("python")`, `which()` will search `PATHEXT` to know that it should look for `python.exe` within the *path* directories. For example, on Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

버전 3.3에 추가.

exception `shutil.`**`Error`**
This exception collects exceptions that are raised during a multi-file operation. For `copytree()`, the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

copytree example

This example is the implementation of the `copytree()` function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except OSError as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except OSError as why:
        # can't copy file access times on Windows
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

Another example that uses the *ignore_patterns()* helper:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the *ignore* argument to add a logging call:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree example

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the onerror callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 Archiving operations

버전 3.2에 추가.

버전 3.5에서 변경: Added support for the *xztar* format.

High-level utilities to create and read compressed and archived files are also provided. They rely on the `zipfile` and `tarfile` modules.

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Create an archive file (such as zip or tar) and return its name.

base_name is the name of the file to create, including the path, minus any format-specific extension. *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), or "xztar" (if the `lzma` module is available).

root_dir is a directory that will be the root directory of the archive, all paths in the archive will be relative to it; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive. *base_dir* must be given relative to *root_dir*. See *Archiving example with base_dir* for how to use *base_dir* and *root_dir* together.

root_dir and *base_dir* both default to the current directory.

If *dry_run* is true, no archive is created, but the operations that would be executed are logged to *logger*.

owner and *group* are used when creating a tar archive. By default, uses the current owner and group.

logger must be an object compatible with **PEP 282**, usually an instance of `logging.Logger`.

The *verbose* argument is unused and deprecated.

shutil.**get_archive_formats**()

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

By default `shutil` provides these formats:

- *zip*: ZIP file (if the `zlib` module is available).
- *tar*: uncompressed tar file.
- *gztar*: gzip'ed tar-file (if the `zlib` module is available).
- *bztar*: bzip2'ed tar-file (if the `bz2` module is available).
- *xztar*: xz'ed tar-file (if the `lzma` module is available).

You can register new formats or provide your own archiver for any existing formats, by using `register_archive_format()`.

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])

Register an archiver for the format *name*.

function is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to `os.curdir`) to start archiving from. Further arguments are passed as keyword arguments: *owner*, *group*, *dry_run* and *logger* (as passed in `make_archive()`).

If given, *extra_args* is a sequence of `(name, value)` pairs that will be used as extra keywords arguments when the archiver callable is used.

description is used by `get_archive_formats()` which returns the list of archivers. Defaults to an empty string.

`shutil.`**`unregister_archive_format`**(*name*)
Remove the archive format *name* from the list of supported formats.

`shutil.`**`unpack_archive`**(*filename*[, *extract_dir*[, *format*]])
Unpack an archive. *filename* is the full path of the archive.

extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

format is the archive format: one of "zip", "tar", "gztar", "bztar", or "xztar". 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.

버전 3.7에서 변경: Accepts a *path-like object* for *filename* and *extract_dir*.

`shutil.`**`register_unpack_format`**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])
Registers an unpack format. *name* is the name of the format and *extensions* is a list of extensions corresponding to the format, like `.zip` for Zip files.

function is the callable that will be used to unpack archives. The callable will receive the path of the archive, followed by the directory the archive must be extracted to.

When provided, *extra_args* is a sequence of `(name, value)` tuples that will be passed as keywords arguments to the callable.

description can be provided to describe the format, and will be returned by the `get_unpack_formats()` function.

`shutil.`**`unregister_unpack_format`**(*name*)
Unregister an unpack format. *name* is the name of the format.

`shutil.`**`get_unpack_formats`**()
Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple `(name, extensions, description)`.

By default `shutil` provides these formats:

- *zip*: ZIP file (unpacking compressed files works only if the corresponding module is available).
- *tar*: uncompressed tar file.
- *gztar*: gzip'ed tar-file (if the `zlib` module is available).
- *bztar*: bzip2'ed tar-file (if the `bz2` module is available).
- *xztar*: xz'ed tar-file (if the `lzma` module is available).

You can register new formats or provide your own unpacker for any existing formats, by using `register_unpack_format()`.

Archiving example

In this example, we create a gzip'ed tar-file archive containing all files found in the `.ssh` directory of the user:

```
>>> 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'
```

The resulting archive contains:

```
$ 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
```

Archiving example with *base_dir*

In this example, similar to the *one above*, we show how to use *make_archive()*, but this time with the usage of *base_dir*. We now have the following directory structure:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
        │   └── please_add.txt
        └── do_not_add.txt
```

In the final archive, `please_add.txt` should be included, but `do_not_add.txt` should not. Therefore we use the following:

```
>>> 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/my_archive.tar'
```

Listing the files in the resulting archive gives us:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.10.3 Querying the size of the output terminal

shutil.**get_terminal_size**(*fallback=(columns, lines)*)

Get the size of the terminal window.

For each of the two dimensions, the environment variable, COLUMNS and LINES respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When COLUMNS or LINES is not defined, which is the common case, the terminal connected to *sys.__stdout__* is queried by invoking *os.get_terminal_size()*.

If the terminal size cannot be successfully queried, either because the system doesn't support querying, or because we are not connected to a terminal, the value given in fallback parameter is used. fallback defaults to (80, 24) which is the default size used by many terminal emulators.

The value returned is a named tuple of type *os.terminal_size*.

See also: The Single UNIX Specification, Version 2, Other Environment Variables.

버전 3.3에 추가.

11.11 macpath — Mac OS 9 path manipulation functions

Source code: Lib/macpath.py

Deprecated since version 3.7, will be removed in version 3.8.

This module is the Mac OS 9 (and earlier) implementation of the *os.path* module. It can be used to manipulate old-style Macintosh pathnames on Mac OS X (or any other platform).

The following functions are available in this module: normcase(), normpath(), isabs(), join(), split(), isdir(), isfile(), walk(), exists(). For other functions available in *os.path* dummy counterparts are available.

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 잘못되었거나 악의적으로 생성된 데이터에 대해 안전하지 않습니다. 신뢰할 수 없거나 인증되지 않은 출처에서 받은 데이터를 절대로 역 피클 하지 마십시오.

[1] 이것을 *marshal* 모듈과 혼동하지 마십시오.

12.1.1 다른 파이썬 모듈과의 관계

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).

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 또는 XDR (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 5가지 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 기본 프로토콜이며, 다른 파이썬 3 버전과의 호환성이 필요한 경우 권장되는 프로토콜입니다.
- 프로토콜 버전 4가 파이썬 3.4에 추가되었습니다. 매우 큰 객체, 더 많은 종류의 객체에 대한 피클링, 일부 데이터 형식 최적화에 대한 지원을 추가합니다. 프로토콜 4에 의해 개선된 사항에 대한 정보는 **PEP 3154**를 참조하십시오.

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**
: 정수, 피클링에 사용되는 기본 프로토콜 버전. *HIGHEST_PROTOCOL* 보다 작을 수 있습니다. 현재 기본 프로토콜은 3인데, 파이썬 3 용으로 설계된 새로운 프로토콜입니다.

pickle 모듈은 피클링 절차를 보다 편리하게 하려고 다음과 같은 함수를 제공합니다:

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*)
: Write the pickled representation of the object *obj* to the open *file object file*. This is equivalent to `Pickler(file, protocol).dump(obj)`.

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.

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*)
Return the pickled representation of the object *obj* as a `bytes` object, instead of writing it to a file.

인자 *protocol* 과 *fix_imports* 는 `dump()` 와 같은 의미입니다.

pickle.**load**(*file*, ***, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)
Read the pickled representation of an object from the open *file object* *file* and return the reconstituted object hierarchy specified therein. This is equivalent to `Unpickler(file).load()`.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled representation of the object are ignored.

인자 *file* 에는 두 가지 메서드가 있어야 합니다, 정수 인자를 받아들이는 read() 메서드와 인자가 없는 readline() 메서드. 두 메서드 모두 바이트열를 반환해야 합니다. 따라서 *file* 은 바이너리 읽기를 위해 열린 디스크 상의 파일, `io.BytesIO` 객체 또는 이 인터페이스를 만족하는 다른 사용자 정의 객체일 수 있습니다.

선택적 키워드 인자는 *fix_imports*, *encoding* 및 *errors* 인데, 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.

pickle.**loads**(*data*, ***, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)
Return the reconstituted object hierarchy of the pickled representation *data* of an object. *data* must be a *bytes-like object*.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled representation of the object are ignored.

선택적 키워드 인자는 *fix_imports*, *encoding* 및 *errors* 인데, 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.

`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 두 개의 클래스를 노출합니다, *Pickler*와 *Unpickler*:

class pickle.**Pickler**(*file*, *protocol=None*, ***, *fix_imports=True*)
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.

dump(*obj*)
Write the pickled representation of *obj* to the open file object given in the constructor.

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.

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.

버전 3.3에 추가.

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.

class pickle.**Unpickler**(*file*, ***, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)
피클 데이터 스트림을 읽는 데 사용될 바이너리 파일을 받아들입니다.

피클의 프로토콜 버전이 자동으로 감지되므로 프로토콜 인자가 필요하지 않습니다.

인자 *file* 에는 두 가지 메서드가 있어야 합니다, 정수 인자를 받아들이는 read() 메서드와 인자가 없는 readline() 메서드. 두 메서드 모두 바이트열을 반환해야 합니다. 따라서 *file* 은 바이너리 읽기를 위해 열린 디스크 상의 파일 객체, *io.BytesIO* 객체 또는 이 인터페이스를 만족하는 다른 사용자 정의 객체일 수 있습니다.

선택적 키워드 인자는 *fix_imports*, *encoding* 및 *errors* 인데, 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.

load()
Read the pickled representation of an object from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein. Bytes past the pickled representation of the object are ignored.

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.

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.

12.1.4 어떤 것이 피클 되고 역 피클 될 수 있을까요?

다음 형을 피클 할 수 있습니다:

- `None`, `True` 와 `False`
- 정수, 실수, 복소수
- 문자열, 바이트열, 바이트 배열(bytearray)
- 피클 가능한 객체만 포함하는 튜플, 리스트, 집합과 딕셔너리
- 모듈의 최상위 수준에서 정의된 함수 (`lambda` 가 아니라 `def` 를 사용하는)
- 모듈의 최상위 수준에서 정의된 내장 함수
- 모듈의 최상위 수준에서 정의된 클래스
- 그런 클래스의 인스턴스 중에서 *__dict__* 나 *__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" 이름 참조로 피클 됨에 유의하십시오.[2] 이것은 함수가 정의된 모듈의 이름과 함께 함수의 이름만 피클 된다는 것을 의미합니다. 함수의 코드도 함수 어트리

[2] 이것이 `lambda` 함수가 pickle 될 수 없는 이유입니다: 모든 `lambda` 함수는 같은 이름을 공유합니다: `<lambda>`.

뷰트도 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 이름 참조로 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:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(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.

[3] 발생하는 예외는 *ImportError* 나 *AttributeError* 일 가능성이 크지만, 그 밖의 다른 것일 수 있습니다.

버전 3.6에서 변경: 파이썬 3.6 이전에는, 프로토콜 2와 3에서 *__getnewargs_ex__()* 대신 *__getnewargs__()* 가 호출되었습니다.

object.**__getstate__**()
클래스는 인스턴스가 피클 되는 방식에 더 많은 영향을 줄 수 있습니다; 클래스가 메서드 *__getstate__()* 를 정의하면, 인스턴스의 딕셔너리 내용 대신, 이 메서드가 호출되고 반환된 객체를 인스턴스의 내용으로 피클 합니다. *__getstate__()* 메서드가 없다면, 인스턴스의 *__dict__* 가 평소와 같이 피클 됩니다.

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: *__getstate__()* 가 거짓 값을 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: At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement `__new__()` to establish such an invariant, as `__init__()` is not called when unpickling an instance.

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 5가 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.

[4] *copy* 모듈은 얕거나 깊은 복사 연산에 이 프로토콜을 사용합니다.

- 선택적으로, 연속적인 항목을 생성하는 이터레이터(시퀀스가 아닙니다). 이 항목들은 `obj.append(item)` 을 사용하거나 한꺼번에 `obj.extend(list_of_items)` 를 사용하여 객체에 추가될 것입니다. 이것은 주로 리스트 서브 클래스에 사용되지만, 적절한 서명을 갖는 `append()` 와 `extend()` 메서드가 있는 한 다른 클래스에서 사용될 수 있습니다. (`append()` 나 `extend()` 중 어느 것이 사용되는지는 어떤 피클 프로토콜 버전이 사용되는가와 추가 할 항목의 수에 따라 달려있으므로 둘 다 지원되어야 합니다.)
- 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.

`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.

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.

The resolution of such persistent IDs is not defined by the *`pickle`* module; it will delegate this resolution to the user-defined methods on the pickler and unpickler, *`persistent_id()`* and *`persistent_load()`* respectively.

To pickle objects that have an external persistent ID, the pickler must have a custom *`persistent_id()`* method that takes an object as an argument and returns either `None` or the persistent ID for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent ID string is returned, the pickler will pickle that object, along with a marker so that the unpickler will recognize it as a persistent ID.

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.

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
```

(다음 페이지에 계속)

[5] 영숫자 문자의 제한은 프로토콜 0에서 지속성 ID가 개행 문자로 구분되기 때문입니다. 따라서 지속성 ID에 개행 문자가 포함되면 결과 피클을 읽을 수 없게 됩니다.

(이전 페이지에서 계속)

```
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    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,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    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, 사설 디스패치 테이블을 갖는 피클러를 만들 수 있습니다.

copyreg 모듈에 의해 관리되는 전역 디스패치 테이블은 copyreg.dispatch_table로 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 같은 디스패치 테이블을 공유합니다. *copyreg* 모듈을 사용하는 동등한 코드는 다음과 같습니다

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

상태 저장 객체 처리

다음은 클래스의 피클 동작을 수정하는 방법을 보여주는 예제입니다. TextReader 클래스는 텍스트 파일을 열고, readline() 메서드가 호출될 때마다 줄 번호와 줄 내용을 반환합니다. TextReader 인스턴스가 피클되면, 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:
    """Print and number lines in a text file."""

    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):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        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 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):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

우리의 역 피클러 작업이 의도한 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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  ...
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.

12.1.7 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.

12.1.8 예제

가장 간단한 코드로, *dump()* 와 *load()* 함수를 사용하십시오.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

다음 예제는 결과로 나온 피클 데이터를 읽습니다.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

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. *pickle*과 *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), *TypeError*가 발생합니다.

copyreg.**pickle**(*type*, *function*, *constructor=None*)
*function*이 *type* 형의 객체에 대한 "환원" 함수로 사용되어야 한다고 선언합니다. *function*는 문자열이나 두 개 또는 세 개의 요소를 포함하는 튜플을 반환해야 합니다.

선택적 *constructor* 매개 변수가 제공되면, 콜러블 객체이며, 피클 할 때 *function*에 의해 반환된 인자의 튜플로 호출될 때 객체를 재구성하는 데 사용할 수 있습니다. *object*가 클래스이거나 *constructor*가 콜러블이 아니면 *TypeError*가 발생합니다.

function 과 *constructor*에서 기대되는 인터페이스에 대한 자세한 내용은 *pickle* 모듈을 참조하십시오. 피클러 객체나 *pickle.Pickler*의 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(object):
...     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 버전 3 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 액세스되면, 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).

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 모든 메서드를 지원합니다. 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.

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 버전 3 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.

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)
pybsddb의 제삼자 `bsddb` 모듈에서는 사용할 수 있지만 다른 데이터베이스 모듈에서는 사용할 수 없는 `first()`, `next()`, `previous()`, `last()` 및 `set_location()`을 노출하는 *Shelf*의 서브 클래스. 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)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
d['xx'] = [0, 1, 2]         # this works as expected, but...
d['xx'].append(3)           # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']              # extracts the copy
temp.append(5)              # mutates the copy
d['xx'] = temp              # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                   # close it
```

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 Sun으로 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. 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.

모든 파이썬 객체 형이 지원되는 것은 아닙니다; 일반적으로, 파이썬의 특정 실행에 무관한 값을 가진 객체만 이 모듈에서 쓰고 읽을 수 있습니다. 다음 형이 지원됩니다: 논릿값, 정수, 부동 소수점 수, 복소수, 문자열, 바이트열, 바이트 배열, 튜플, 리스트, 집합, frozenset, 딕셔너리 및 코드 객체, 여기서 튜플, 리스트, 집합, frozenset 및 딕셔너리는 포함된 값이 자체적으로 지원될 때만 지원됩니다. 싱글톤 *None*, *Ellipsis* 및 *StopIteration*도 마샬과 역 마샬 될 수 있습니다. 형식 *version*이 3보다 작으면, 재귀적인 리스트, 집합 및 딕셔너리를 기록할 수 없습니다 (아래를 참조하십시오).

파일을 읽고 쓰는 함수는 물론 바이트열류 객체에서 작동하는 함수도 있습니다.

모듈은 다음 함수를 정의합니다:

[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.

marshal.**dump**(*value*, *file*[, *version*])
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.

version 인자는 dump가 사용해야 하는 데이터 형식을 나타냅니다 (아래를 참조하십시오).

marshal.**load**(*file*)
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. 파일은 읽을 수 있는 바이너리 파일 이어야 합니다.

참고: 지원하지 않는 형을 포함하는 객체가 *dump()*로 마샬 되었으면, *load()*는 역 마샬이 불가능한 형을 None으로 치환합니다.

---

marshal.**dumps**(*value*[, *version*])
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.

version 인자는 dumps가 사용해야 하는 데이터 형식을 나타냅니다 (아래를 참조하십시오).

marshal.**loads**(*bytes*)
바이트열류 객체를 값으로 변환합니다. 유효한 값이 없으면 *EOFError*, *ValueError* 또는 *TypeError*를 발생시킵니다. 입력의 여분의 바이트는 무시됩니다.

또한, 다음 상수가 정의됩니다:

marshal.**version**
모듈이 사용하는 형식을 나타냅니다. 버전 0은 역사적인 형식이고, 버전 1은 인턴 된 문자열을 공유하고, 버전 2는 부동 소수점 숫자에 바이너리 형식을 사용합니다. 버전 3에서는 객체 인스턴스 화와 재귀에 대한 지원이 추가되었습니다. 현재 버전은 4입니다.

12.5 dbm — 유닉스 "데이터베이스" 인터페이스

소스 코드: Lib/dbm/__init__.py

---

*dbm*은 DBM 데이터베이스 변형에 대한 일반 인터페이스입니다 — *dbm.gnu* 또는 *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.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(''); 또는 필요한 모듈 이름을 포함하는 문자열, 가령 'dbm.ndbm'이나 'dbm.gnu'.

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)

데이터베이스 파일 *file*을 열고 해당 객체를 반환합니다.

데이터베이스 파일이 이미 존재하면, *whichdb()* 함수를 사용하여 유형을 판별하고 적절한 모듈이 사용됩니다; 존재하지 않으면, 위에 나열된 것 중 임포트 할 수 있는 첫 번째 모듈이 사용됩니다.

선택적 *flag* 인자는 다음과 같은 것이 될 수 있습니다:

값	의미
`'r'`	읽기 전용으로 기존 데이터베이스 열기 (기본값)
`'w'`	읽고 쓰기 위해 기존 데이터베이스 열기
`'c'`	읽고 쓰기 위해 데이터베이스를 열고, 존재하지 않으면 만들기
`'n'`	읽고 쓰기 위해 항상 새로운 빈 데이터베이스를 만들기

선택적 *mode* 인자는 파일의 유닉스 모드이며, 데이터베이스를 만들 때만 사용됩니다. 기본값은 8진수 `0o666`입니다 (그리고 현재 umask에 의해 수정됩니다).

*open()*이 반환한 객체는 딕셔너리와 같은 기본 기능을 지원합니다; 키와 해당 값을 저장, 조회 및 삭제할 수 있으며, `get()`과 `setdefault()`뿐만 아니라 `in` 연산자와 `keys()` 메서드도 사용할 수 있습니다.

버전 3.2에서 변경: 이제 모든 데이터베이스 모듈에서 `get()`과 `setdefault()`를 사용할 수 있습니다.

키와 값은 항상 바이트열로 저장됩니다. 이는 문자열이 사용될 때 저장되기 전에 기본 인코딩으로 묵시적으로 변환됨을 의미합니다.

이 객체는 `with` 문에서도 사용되도록 지원해서, 완료될 때 자동으로 닫힙니다.

버전 3.4에서 변경: *open()*이 반환한 객체에 컨텍스트 관리 프로토콜에 대한 기본 지원을 추가했습니다.

다음 예제는 일부 호스트 명과 해당 제목을 기록한 다음, 데이터베이스의 내용을 인쇄합니다:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

더 보기:

모듈 *shelve* 문자열이 아닌 데이터를 저장하는 지속성 모듈.

개별 서브 모듈은 다음 섹션에서 설명합니다.

12.5.1 dbm.gnu — GNU의 dbm 재해석

소스 코드: Lib/dbm/gnu.py

이 모듈은 *dbm* 모듈과 매우 유사하지만, GNU 라이브러리 gdbm을 대신 사용하여 추가 기능을 제공합니다. *dbm.gnu*와 *dbm.ndbm*으로 만든 파일 형식은 서로 호환되지 않음에 유의하십시오.

dbm.gnu 모듈은 GNU DBM 라이브러리에 대한 인터페이스를 제공합니다. dbm.gnu.gdbm 객체는 키와 값이 저장되기 전에 항상 바이트열로 변환된다는 점을 제외하고는 매핑(딕셔너리)처럼 동작합니다. gdbm 객체를 인쇄해도 키와 값이 인쇄되지 않으며, items() 와 values() 메서드는 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*[, *mode*]])
gdbm 데이터베이스를 열고 gdbm 객체를 반환합니다. *filename* 인자는 데이터베이스 파일의 이름입니다.

선택적 *flag* 인자는 다음과 같은 것이 될 수 있습니다:

값	의미
'r'	읽기 전용으로 기존 데이터베이스 열기 (기본값)
'w'	읽고 쓰기 위해 기존 데이터베이스 열기
'c'	읽고 쓰기 위해 데이터베이스를 열고, 존재하지 않으면 만들기
'n'	읽고 쓰기 위해 항상 새로운 빈 데이터베이스를 만들기

데이터베이스를 여는 방법을 제어하기 위해 다음과 같은 추가 문자가 flag에 추가될 수 있습니다:

값	의미
'f'	데이터베이스를 빠른 모드로 엽니다. 데이터베이스로의 쓰기는 동기화되지 않습니다.
's'	동기화 모드. 이것은 데이터베이스 변경 사항이 파일에 즉시 기록되도록 합니다.
'u'	데이터베이스를 잠그지 않습니다.

모든 플래그가 모든 버전의 gdbm에서 유효한 것은 아닙니다. 모듈 상수 open_flags는 지원되는 플래그 문자의 문자열입니다. 유효하지 않은 플래그가 지정되면 *error* 예외가 발생합니다.

선택적 *mode* 인자는 파일의 유닉스 모드이며, 데이터베이스를 만들어야 할 때만 사용됩니다. 기본값은 8진수 0o666입니다.

딕셔너리와 유사한 메서드 외에도, gdbm 객체에는 다음과 같은 메서드가 있습니다:

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 db의 모든 키를 인쇄합니다:

```
k = db.firstkey()
while k != 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.

12.5.2 dbm.ndbm — ndbm 기반 인터페이스

소스 코드: Lib/dbm/ndbm.py

---

dbm.ndbm 모듈은 유닉스 "(n) dbm" 라이브러리에 대한 인터페이스를 제공합니다. Dbm 객체는 키와 값이 항상 바이트열로 저장된다는 점을 제외하고는, 매핑(딕셔너리)처럼 동작합니다. dbm 객체를 인쇄해도 키와 값이 인쇄되지 않으며, items() 와 values() 메서드는 지원되지 않습니다.

이 모듈은 "고전적인" ndbm 인터페이스나 GNU GDBM 호환 인터페이스로 사용할 수 있습니다. 유닉스에서, **configure** 스크립트는 이 모듈 빌드를 단순화하기 위해 적절한 헤더 파일을 찾습니다.

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*[, *mode*]])
dbm 데이터베이스를 열고 ndbm 객체를 반환합니다. *filename* 인자는 데이터베이스 파일의 이름입니다 (.dir이나 .pag 확장자는 없습니다).

선택적 *flag* 인자는 다음 값 중 하나여야 합니다:

값	의미
'r'	읽기 전용으로 기존 데이터베이스 열기 (기본값)
'w'	읽고 쓰기 위해 기존 데이터베이스 열기
'c'	읽고 쓰기 위해 데이터베이스를 열고, 존재하지 않으면 만들기
'n'	읽고 쓰기 위해 항상 새로운 빈 데이터베이스를 만들기

선택적 *mode* 인자는 파일의 유닉스 모드이며, 데이터베이스를 만들 때만 사용됩니다. 기본값은 8진수 0o666입니다 (그리고 현재 umask에 의해 수정됩니다).

딕셔너리와 유사한 메서드 외에도, ndbm 객체는 다음 메서드를 제공합니다:

ndbm.**close**()
ndbm 데이터베이스를 닫습니다.

12.5.3 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) 딕셔너리와 유사한 인터페이스를 제공합니다. *dbm.gnu*와 같은 다른 모듈과 달리, 외부 라이브러리가 필요하지 않습니다. 다른 지속성 매핑처럼, 키와 값은 항상 바이트열로 저장됩니다.

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*[, *mode*]])
dumbdbm 데이터베이스를 열고 dumbdbm 객체를 반환합니다. *filename* 인자는 데이터베이스 파일의 베이스 이름입니다 (특정 확장자는 없습니다). dumbdbm 데이터베이스가 만들어질 때, `.dat`와 `.dir` 확장자를 가진 파일이 만들어집니다.

The optional *flag* argument supports only the semantics of `'c'` and `'n'` values. Other values will default to database being always opened for update, and will be created if it does not exist.

선택적 *mode* 인자는 파일의 유닉스 모드이며, 데이터베이스를 만들 때만 사용됩니다. 기본값은 8진수 `0o666`입니다 (그리고 현재 umask에 의해 수정됩니다).

> **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: flag에 `'n'` 값이 있으면, *open()*은 항상 새 데이터베이스를 만듭니다.

Deprecated since version 3.6, will be removed in version 3.8: Creating database in `'r'` and `'w'` modes. Modifying database in `'r'` mode.

collections.abc.MutableMapping 클래스가 제공하는 메서드 외에도, dumbdbm 객체는 다음 메서드를 제공합니다:

dumbdbm.**sync**()
디스크 상의 디렉터리와 데이터 파일을 동기화합니다. 이 메서드는 `Shelve.sync()` 메서드에 의해 호출됩니다.

dumbdbm.**close**()
dumbdbm 데이터베이스를 닫습니다.

12.6 sqlite3 — SQLite 데이터베이스용 DB-API 2.0 인터페이스

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.

sqlite3 모듈은 Gerhard Häring이 썼습니다. **PEP 249**에서 설명하는 DB-API 2.0 명세를 준수하는 SQL 인터페이스를 제공합니다.

모듈을 사용하려면, 먼저 데이터베이스를 나타내는 *Connection* 객체를 만들어야 합니다. 여기서 데이터는 `example.db` 파일에 저장됩니다

```
import sqlite3
conn = sqlite3.connect('example.db')
```

특수 이름 `:memory:`를 제공하여 램에 데이터베이스를 만들 수도 있습니다.

일단 *Connection*를 얻으면, *Cursor* 객체를 만들고 *execute()* 메서드를 호출하여 SQL 명령을 수행할 수 있습니다.:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

저장한 데이터는 영구적이며 이후 세션에서 사용할 수 있습니다:

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

일반적으로 SQL 연산은 파이썬 변수의 값을 사용해야 합니다. 파이썬의 문자열 연산을 사용하여 질의를 조합해서는 안 됩니다. 그렇게 하는 것은 안전하지 않기 때문입니다; 프로그램이 SQL 인젝션 공격에 취약하게 만듭니다(잘못될 수 있는 유머러스한 예를 보려면 https://xkcd.com/327/ 를 참조하십시오).

대신 DB-API의 매개 변수 치환을 사용하십시오. 값을 사용하고자 할 때마다 `?`를 자리 표시자로 넣은 다음, 커서의 *execute()* 메서드에 두 번째 인자로 값들의 튜플을 제공하십시오. (다른 데이터베이스 모듈은 다른 자리 표시자를 사용할 수 있습니다, 가령 `%s` 나 `:1`.) 예를 들면:

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

SELECT 문을 실행한 후 데이터를 꺼내려면, 커서를 이터레이터로 취급하거나, 커서의 *fetchone()* 메서드를 호출하여 일치하는 단일 행을 꺼내거나, *fetchall()*를 호출하여 일치하는 행의 리스트를 가져올 수 있습니다.

이 예제는 이터레이터 방식을 사용합니다:

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

더 보기:

https://github.com/ghaering/pysqlite pysqlite 웹 페이지 – sqlite3은 "pysqlite"라는 이름으로 외부에서 개발되었습니다.

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 모듈 함수와 상수

sqlite3.**version**
이 모듈의 버전 번호(문자열). SQLite 라이브러리의 버전이 아닙니다.

sqlite3.**version_info**
이 모듈의 버전 번호(정수들의 튜플). SQLite 라이브러리의 버전이 아닙니다.

sqlite3.**sqlite_version**
런타임 SQLite 라이브러리의 버전 번호(문자열).

sqlite3.**sqlite_version_info**
런타임 SQLite 라이브러리의 버전 번호(정수들의 튜플).

sqlite3.**PARSE_DECLTYPES**
이 상수는 *connect()* 함수의 *detect_types* 매개 변수에 사용됩니다.

이것을 설정하면 *sqlite3* 모듈은 반환되는 각 열에 대해 선언된 형을 구문 분석합니다. 선언된 형의 첫 번째 단어를 구문 분석합니다, 즉 "integer primary key"에서는 "integer"를, "number (10)"에서는 "number"

를 구문 분석합니다. 그런 다음 해당 열에 대해, 변환기 딕셔너리를 조사하고 그 형에 대해 등록된 변환기 함수를 사용합니다.

sqlite3.**PARSE_COLNAMES**
이 상수는 *connect()* 함수의 *detect_types* 매개 변수에 사용됩니다.

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that 'mytype' is the type of the column. It will try to find an entry of 'mytype' in the converters dictionary and then use the converter function found there to return the value. The column name found in *Cursor.description* does not include the type, i. e. if you use something like `'as "Expiration date [datetime]"'` in your SQL, then we will parse out everything until the first `'['` for the column name and strip the preceeding space: the column name would simply be "Expiration date".

sqlite3.**connect**(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, *uri*])
SQLite 데이터베이스 파일 *database*에 대한 연결을 엽니다. 사용자 정의 *factory*가 주어지지 않는 한, 기본적으로 *Connection* 객체를 반환합니다.

*database*는 열릴 데이터베이스 파일의 경로명(절대 혹은 현재 작업 디렉터리에 대한 상대)을 제공하는 경로류 객체입니다. `":memory:"`를 사용하여 디스크 대신 램(RAM)에 있는 데이터베이스에 대한 데이터베이스 연결을 열 수 있습니다.

데이터베이스가 여러 연결을 통해 액세스 되고, 프로세스 중 하나가 데이터베이스를 수정할 때, 해당 트랜잭션이 커밋될 때까지 SQLite 데이터베이스가 잠깁니다. *timeout* 매개 변수는 예외를 일으키기 전에 잠금이 해제되기를 연결이 기다려야 하는 시간을 지정합니다. timeout 매개 변수의 기본값은 5.0(5초)입니다.

isolation_level 매개 변수는 *Connection* 객체의 *isolation_level* 프로퍼티를 참조하십시오.

SQLite는 기본적으로 TEXT, INTEGER, REAL, BLOB 및 NULL 형만 지원합니다. 다른 형을 사용하려면 직접 지원을 추가해야 합니다. *detect_types* 매개 변수와 모듈 수준 *register_converter()* 함수로 등록된 사용자 정의 **변환기**를 사용하면 쉽게 할 수 있습니다.

*detect_types*의 기본값은 0입니다(즉, 형 감지가 없습니다). *PARSE_DECLTYPES*와 *PARSE_COLNAMES*의 조합으로 설정하여 형 감지를 켤 수 있습니다.

기본적으로 *check_same_thread*는 *True*며, 만들고 있는 스레드 만 이 연결을 사용할 수 있습니다. *False*로 설정하면 반환된 연결을 여러 스레드에서 공유할 수 있습니다. 여러 스레드에서 같은 연결을 사용할 때, 데이터 손상을 피하려면 쓰기 연산을 사용자가 직렬화해야 합니다.

기본적으로, *sqlite3* 모듈은 connect 호출에 *Connection* 클래스를 사용합니다. 그러나, *Connection* 클래스의 서브 클래스를 만들고 *factory* 매개 변수에 클래스를 제공하면 *connect()*가 그 클래스를 사용하게 할 수 있습니다.

자세한 내용은 이 설명서의 섹션 *SQLite 와 파이썬 형*을 참조하십시오.

sqlite3 모듈은 내부적으로 SQL 구문 분석 오버헤드를 피하고자 명령문 캐시를 사용합니다. 연결에 대해 캐시되는 명령문의 수를 명시적으로 설정하려면, *cached_statements* 매개 변수를 설정할 수 있습니다. 현재 구현된 기본값은 100개의 명령문을 캐시 하는 것입니다.

*uri*가 참이면 *database*는 URI로 해석됩니다. 이렇게 하면 옵션을 지정할 수 있습니다. 예를 들어, 읽기 전용 모드로 데이터베이스를 열려면 다음과 같이 할 수 있습니다:

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

인식되는 옵션 목록을 포함하여, 이 기능에 대한 자세한 내용은 SQLite URI documentation에서 찾을 수 있습니다.

버전 3.4에서 변경: *uri* 매개 변수가 추가되었습니다.

버전 3.7에서 변경: *database*는 이제 문자열뿐만 아니라 경로류 객체 일 수도 있습니다.

sqlite3.**register_converter**(*typename*, *callable*)
데이터베이스의 바이트열을 사용자 정의 파이썬 형으로 변환할 수 있는 콜러블을 등록합니다. 콜러블은 형 *typename* 인 모든 데이터베이스 값에 대해 호출됩니다. 형 감지 작동 방식에 대해서는 *connect()* 함수의 매개 변수 *detect_types*를 참고하십시오. *typename*과 질의의 형 이름은 대/소문자를 구분하지 않고 일치시킴에 유의하십시오.

sqlite3.**register_adapter**(*type*, *callable*)
사용자 정의 파이썬 형 *type*을 SQLite의 지원되는 형 중 하나로 변환할 수 있는 콜러블을 등록합니다. 콜러블 *callable*은 단일 매개 변수로 파이썬 값을 받아들이고 다음 형들의 값을 반환해야 합니다: int, float, str 또는 bytes.

sqlite3.**complete_statement**(*sql*)
문자열 *sql*에 세미콜론으로 끝나는 하나 이상의 완전한 SQL 문이 포함되어 있으면 *True*를 반환합니다. SQL이 문법적으로 올바른지 확인하지는 않습니다. 닫히지 않은 문자열 리터럴이 없고 명령문이 세미콜론으로 끝나는지만 확인합니다.

이것은 다음 예제와 같이, SQLite 용 셸을 만드는데 사용할 수 있습니다:

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""

con.close()
```

sqlite3.**enable_callback_tracebacks**(*flag*)
기본적으로 사용자 정의 함수, 집계(aggregates), 변환기, 인가(authorizer) 콜백 등에서는 트레이스백을 얻지 못합니다. 디버깅하려면 *flag*를 True로 설정하여 이 함수를 호출할 수 있습니다. 그러면, sys.stderr로 콜백의 트레이스백을 얻게 됩니다. 기능을 다시 비활성화하려면 *False*를 사용하십시오.

12.6.2 Connection 객체

class sqlite3.**Connection**

SQLite 데이터베이스 연결에는 다음과 같은 어트리뷰트와 메서드가 있습니다:

isolation_level

현재의 기본 격리 수준을 가져오거나 설정합니다. 자동 커밋 모드를 뜻하는 *None* 이나 "DEFERRED", "IMMEDIATE" 또는 "EXCLUSIVE" 중 하나입니다. 자세한 설명은 트랜잭션 제어 절을 참조하십시오.

in_transaction

트랜잭션이 활성화 상태면(커밋되지 않은 변경 사항이 있으면) *True*, 그렇지 않으면 *False*. 읽기 전용 어트리뷰트.

버전 3.2에 추가.

cursor(*factory=Cursor*)

cursor 메서드는 단일 선택적 매개 변수 *factory*를 받아들입니다. 제공되면, 이것은 *Cursor* 나 그 서브 클래스의 인스턴스를 반환하는 콜러블이어야 합니다.

commit()

이 메서드는 현재 트랜잭션을 커밋합니다. 이 메서드를 호출하지 않으면, 마지막 commit() 호출 이후에 수행한 작업은 다른 데이터베이스 연결에서 볼 수 없습니다. 데이터베이스에 기록한 데이터가 왜 보이지 않는지 궁금하면, 이 메서드를 호출하는 것을 잊지 않았는지 확인하십시오.

rollback()

이 메서드는 마지막 *commit()* 호출 이후의 데이터베이스에 대한 모든 변경 사항을 되돌립니다.

close()

데이터베이스 연결을 닫습니다. 자동으로 *commit()*을 호출하지 않음에 유의하십시오. *commit()*를 먼저 호출하지 않고 데이터베이스 연결을 닫으면 변경 사항이 손실됩니다!

execute(*sql*[, *parameters*])

이것은 비표준 바로 가기인데, *cursor()* 메서드를 호출하여 커서 객체를 만들고, 지정된 *parameters*를 사용하여 커서의 *execute()* 메서드를 호출한 다음, 커서를 반환합니다.

executemany(*sql*[, *parameters*])

이것은 비표준 바로 가기인데, *cursor()* 메서드를 호출하여 커서 객체를 만들고, 지정된 *parameters*를 사용하여 커서의 *executemany()* 메서드를 호출한 다음, 커서를 반환합니다.

executescript(*sql_script*)

이것은 비표준 바로 가기인데, *cursor()* 메서드를 호출하여 커서 객체를 만들고, 지정된 *sql_script*를 사용하여 커서의 *executescript()* 메서드를 호출한 다음, 커서를 반환합니다.

create_function(*name*, *num_params*, *func*)

나중에 함수 이름 *name*으로 SQL 문에서 사용할 수 있는 사용자 정의 함수를 만듭니다. *num_params*는 함수가 받아들이는 매개 변수의 수입니다 (*num_params*가 -1이면 함수는 임의의 인자를 취할 수 있습니다). *func*는 SQL 함수로 호출되는 파이썬 콜러블입니다.

함수는 SQLite가 지원하는 모든 형을 반환할 수 있습니다: bytes, str, int, float 및 None.

예:

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])

con.close()
```

create_aggregate(*name*, *num_params*, *aggregate_class*)
사용자 정의 집계(aggregate) 함수를 만듭니다.

매개 변수의 수 *num_params*(*num_params*가 -1이면 함수는 임의의 인자를 취할 수 있습니다)를 받아들이며, 집계 클래스는 `step` 메서드와 집계의 최종 결과를 반환하는 `finalize` 메서드를 구현해야 합니다.

`finalize` 메서드는 SQLite가 지원하는 모든 형을 반환할 수 있습니다: bytes, str, int, float 및 `None`.

예:

```
import sqlite3

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.cursor()
cur.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()
```

create_collation(*name*, *callable*)
지정된 *name* 과 *callable*로 정렬법(collation)을 만듭니다. 콜러블에는 두 개의 문자열 인자가 전달됩니다. 첫째가 둘째보다 작은 순서면 -1, 같은 순서면 0, 첫째가 둘째보다 큰 순서면 1을 반환 해야 합니다. 이것은 정렬(SQL의 ORDER BY)을 제어하므로, 여러분의 비교는 다른 SQL 연산에 영향을 주지 않습니다.

콜러블 객체는 보통 UTF-8로 인코딩된 파이썬 바이트열로 매개 변수를 가져옵니다.

다음 예제는 "잘못된 방법"으로 정렬하는 사용자 정의 정렬법을 보여줍니다:

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.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()
```

정렬법을 제거하려면 callable에 None을 사용해서 create_collation를 호출하십시오:

```
con.create_collation("reverse", None)
```

interrupt()
연결에서 실행 중일 수 있는 모든 질의를 중단하려면, 이 메서드를 다른 스레드에서 호출할 수 있습니다. 그러면 질의가 중단되고 호출자는 예외를 받습니다.

set_authorizer(*authorizer_callback*)
이 루틴은 콜백을 등록합니다. 콜백은 데이터베이스의 테이블 열에 액세스할 때마다 호출됩니다. 콜백은 액세스가 허용되면 SQLITE_OK를 반환하고, 전체 SQL 문을 에러를 일으키며 중단해야 하면 SQLITE_DENY를, 열을 NULL 값으로 처리하려면 SQLITE_IGNORE를 반환해야 합니다. 이 상수들은 *sqlite3* 모듈에 있습니다.

콜백의 첫 번째 인자는 어떤 종류의 연산이 인가받으려 하는지를 나타냅니다. 두 번째와 세 번째 인자는 첫 번째 인자에 따라 인자이거나 *None*이 됩니다. 네 번째 인자는 해당하면 데이터베이스 이름("main", "temp" 등) 입니다. 다섯 번째 인자는 액세스 시도를 담당하는 가장 안쪽의 트리거나 뷰의 이름이거나, 이 액세스 시도가 입력 SQL 코드에서 직접 발생했으면 *None*입니다.

첫 번째 인자에 가능한 값과 첫 번째 인자에 의존하는 두 번째 및 세 번째 인자의 의미에 대해서는 SQLite 문서를 참조하십시오. 필요한 모든 상수는 *sqlite3* 모듈에 있습니다.

set_progress_handler(*handler*, *n*)
이 루틴은 콜백을 등록합니다. 콜백은 SQLite 가상 머신의 매 *n*개의 명령어마다 호출됩니다. 장시간 실행되는 작업 중에 SQLite로부터 호출되기를 원할 때 유용합니다, 예를 들어 GUI를 갱신하는데 사용할 수 있습니다.

이전에 설치된 모든 진행 처리기를 지우려면 *handler*로 *None*을 사용하여 메서드를 호출하십시오.

처리기 함수에서 0이 아닌 값을 반환하면 현재 실행 중인 질의가 종료되고 *OperationalError* 예외가 발생합니다.

set_trace_callback(*trace_callback*)
SQLite 백 엔드가 실제로 실행하는 각 SQL 문마다 호출할 *trace_callback*을 등록합니다.

콜백에 전달되는 유일한 인자는 실행 중인 문장(문자열) 입니다. 콜백의 반환 값은 무시됩니다. 백 엔드는 *Cursor.execute()* 메서드에 전달된 명령문만 실행하는 것은 아님에 유의하시기 바랍니다. 다른 소스로는 파이썬 모듈의 트랜잭션 관리와 현재 데이터베이스에 정의된 트리거의 실행이 있습니다.

*None*을 *trace_callback*로 전달하면 추적 콜백을 비활성화합니다.

버전 3.3에 추가.

enable_load_extension(*enabled*)
이 루틴은 SQLite 엔진이 공유 라이브러리에서 SQLite 확장을 로드하는 것을 허용/불허합니다.

SQLite 확장은 새 함수, 집계 또는 완전히 새로운 가상 테이블 구현을 정의할 수 있습니다. 잘 알려진 확장 중 하나는 SQLite와 함께 배포되는 전체 텍스트 검색 확장입니다.

로드 가능한 확장은 기본적으로 비활성화되어 있습니다.[1]를 보세요.

버전 3.2에 추가.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
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)

con.close()
```

load_extension(*path*)
이 루틴은 공유 라이브러리에서 SQLite 확장을 로드합니다. 이 루틴을 사용하려면 먼저 *enable_load_extension()*로 확장 로드를 활성화해야 합니다.

로드 가능한 확장은 기본적으로 비활성화되어 있습니다.[1]를 보세요.

버전 3.2에 추가.

row_factory
이 어트리뷰트를 커서와 원본 행을 튜플로 받아들이고 실제 결과 행을 반환하는 콜러블로 변경할 수 있습니다. 이렇게 하면, 이름으로 열을 액세스할 수 있는 객체를 반환하는 것과 같이, 결과를 반환하는 더 고급 방식을 구현할 수 있습니다.

예:

[1] 기본적으로 sqlite3 모듈은 로드 가능한 확장을 지원하도록 빌드되지 않습니다. 일부 플랫폼(특히 맥 OS X)에는 이 기능 없이 컴파일된 SQLite 라이브러리가 있기 때문입니다. 로드 가능한 확장 지원을 받으려면, configure에 –enable-loadable-sqlite-extensions를 전달해야 합니다.

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])

con.close()
```

튜플을 반환하는 것으로 충분하지 않고 열에 대한 이름 기반 액세스를 원하면, *row_factory*를 고도로 최적화된 *sqlite3.Row* 형으로 설정하는 것을 고려해야 합니다. *Row*는 메모리 오버헤드가 거의 없이 열에 대해 인덱스 기반과 대소 문자를 구분하지 않는 이름 기반 액세스를 제공합니다. 아마도 여러분 자신의 사용자 정의 딕셔너리 기반 접근법이나 심지어 db_row 기반 해법보다 더 좋을 것입니다.

text_factory

이 어트리뷰트를 사용하면 TEXT 데이터형에 대해 반환되는 객체를 제어할 수 있습니다. 기본적으로, 이 어트리뷰트는 *str*로 설정되고 *sqlite3* 모듈은 TEXT에 대해 유니코드 객체를 반환합니다. 대신 바이트열을 반환하려면, *bytes*로 설정할 수 있습니다.

하나의 바이트열 매개 변수를 받아들이고 결과 객체를 반환하는 다른 콜러블 객체로 설정할 수도 있습니다.

예시를 위해 다음 예제 코드를 참조하십시오:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
assert row[0] == "barfoo"

con.close()
```

total_changes
데이터베이스 연결이 열린 후 수정, 삽입 또는 삭제된 데이터베이스 행의 총수를 반환합니다.

iterdump()
SQL 텍스트 형식으로 데이터베이스를 덤프하는 이터레이터를 반환합니다. 나중에 복원할 수 있도록 메모리 데이터베이스를 저장할 때 유용합니다. 이 함수는 **sqlite3** 셸의 `.dump` 명령과 같은 기능을 제공합니다.

예:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

backup(*target*, ***, *pages=0*, *progress=None*, *name="main"*, *sleep=0.250*)
이 메서드는 SQLite 데이터베이스의 백업을 만드는데, 다른 클라이언트가 액세스하고 있거나 같은 연결로 동시에 액세스하고 있어도 됩니다. 복사본은 필수 인자 *target*에 기록되며, 이것은 다른 *Connection* 인스턴스여야 합니다.

기본적으로, 또는 *pages*가 `0` 이나 음의 정수이면, 전체 데이터베이스가 단일 단계로 복사됩니다; 그렇지 않으면 이 메서드는 한 번에 최대 *pages* 페이지만큼 복사하는 루프를 수행합니다.

*progress*가 지정되면, `None` 또는 매 이터레이션마다 세 개의 정수 인자로 실행되는 콜러블 객체여야 합니다. 세 인자는 각각 직전 이터레이션의 상태 *(status)*, 아직 복사해야 할 남은 *(remaining)* 페이지 수, 전체 *(total)* 페이지 수입니다.

name 인자는 복사할 데이터베이스 이름을 지정합니다: main 데이터베이스를 나타내는 `"main"`, 기본값, 임시 데이터베이스를 나타내는 `"temp"` 또는 첨부된 데이터베이스를 위한 `ATTACH DATABASE` 문에서 `AS` 키워드 뒤에 지정된 이름을 포함하는 문자열이어야 합니다.

sleep 인자는 남은 페이지를 백업하는 연속적인 시도 사이에서 잠잘 시간을 초 단위로 지정하며, 정수 또는 부동 소수점 값으로 지정할 수 있습니다.

예제 1, 기존 데이터베이스를 다른 데이터베이스로 복사:

```
import sqlite3

def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

con = sqlite3.connect('existing_db.db')
bck = sqlite3.connect('backup.db')
with bck:
    con.backup(bck, pages=1, progress=progress)
bck.close()
con.close()
```

예제 2, 기존 데이터베이스를 임시 복사본으로 복사:

```
import sqlite3

source = sqlite3.connect('existing_db.db')
dest = sqlite3.connect(':memory:')
source.backup(dest)
```

가용성: SQLite 3.6.11 이상

버전 3.7에 추가.

12.6.3 Cursor 객체

class sqlite3.**Cursor**

Cursor 인스턴스에는 다음과 같은 어트리뷰트와 메서드가 있습니다.

execute(*sql*[, *parameters*])

SQL 문을 실행합니다. SQL 문을 매개 변수화(즉, SQL 리터럴 대신 자리 표시자) 할 수 있습니다. *sqlite3* 모듈은 두 가지 자리 표시자를 지원합니다: 물음표(qmark 스타일)와 이름있는 자리 표시자(named 스타일).

다음은 두 스타일의 예입니다:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who":␣
↪who, "age": age})

print(cur.fetchone())

con.close()
```

*execute()*는 단일 SQL 문만 실행합니다. 하나 이상의 명령문을 실행하려고 하면 *Warning*이 발생합니다. 하나의 호출로 여러 SQL 문을 실행하려면 *executescript()*를 사용하십시오.

executemany(*sql*, *seq_of_parameters*)

시퀀스 *seq_of_parameters*에 있는 모든 매개 변수 시퀀스나 매핑에 대해 SQL 명령을 실행합니다. *sqlite3* 모듈은 시퀀스 대신 매개 변수를 산출하는 이터레이터도 허용합니다.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

다음은 제너레이터를 사용하는 간단한 예입니다:

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

executescript(*sql_script*)

이것은 한 번에 여러 SQL 문을 실행하기 위한 비표준 편의 메서드입니다. 먼저 COMMIT 문을 실행한 다음, 매개 변수로 가져온 SQL 스크립트를 실행합니다.

*sql_script*는 *str*의 인스턴스가 될 수 있습니다.

예:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
con.close()
```

fetchone()
질의 결과 집합의 다음 행을 가져옵니다. 단일 시퀀스를 반환하거나, 데이터가 더 없을 때 *None*을 반환합니다.

fetchmany(*size=cursor.arraysize*)
질의 결과의 다음 행 집합을 가져와서, 리스트를 반환합니다. 행이 더 없으면 빈 목록이 반환됩니다.

호출 당 가져오는 행의 수는 *size* 매개 변수로 지정됩니다. 지정되어 않으면, 커서의 arraysize가 가져올 행의 수를 결정합니다. 이 메서드는 size 매개 변수가 나타내는 수만큼의 행을 가져오려고 해야 합니다. 지정된 수의 행이 없어서 이것이 가능하지 않다면, 더 적은 행이 반환될 수 있습니다.

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()
질의 결과의 모든 (남은) 행을 가져와서 리스트를 반환합니다. 커서의 arraysize 어트리뷰트는 이 연산의 성능에 영향을 줄 수 있습니다. 행이 없으면 빈 리스트가 반환됩니다.

close()
(__del__이 호출 될 때가 아니라) 지금 커서를 닫습니다.

이 시점부터는 커서를 사용할 수 없습니다; 커서로 어떤 연산이건 시도하면 *ProgrammingError* 예외가 발생합니다.

rowcount
sqlite3 모듈의 *Cursor* 클래스가 이 어트리뷰트를 구현하지만, "영향을 받는 행"/"선택된 행"의 판단을 위한 데이터베이스 엔진 자체 지원은 기이합니다.

executemany() 문에서, 수정 횟수는 *rowcount*에 합산됩니다.

파이썬 DB API 스펙에 따라, *rowcount* 어트리뷰트는 커서에서 executeXX() 가 수행되지 않았거나 마지막 연산의 행 개수가 인터페이스에 의해 결정되지 않는 경우 -1입니다. 이런 경우는 SELECT 문을 포함하는데, 모든 행을 가져올 때까지 질의가 생성 한 행 수를 결정할 수 없기 때문입니다.

3.6.5 이전의 SQLite 버전에서는, 조건 없이 DELETE FROM table을 하면 *rowcount*가 0으로 설정됩니다.

lastrowid
이 읽기 전용 어트리뷰트는 마지막으로 수정된 행의 rowid를 제공합니다. *execute()* 메서드를 사용하여 INSERT 나 REPLACE 문을 실행했을 때만 설정됩니다. INSERT 나 REPLACE 이외의 연산이나 *executemany()* 가 호출될 때, *lastrowid*는 *None*으로 설정됩니다.

INSERT 나 REPLACE 문이 삽입에 실패하면, 이전의 성공적인 rowid가 반환됩니다.

버전 3.6에서 변경: REPLACE 문에 대한 지원이 추가되었습니다.

arraysize
*fetchmany()*에 의해 반환되는 행의 수를 제어하는 읽기/쓰기 어트리뷰트. 기본값은 1입니다. 이는 호출 당 하나의 행을 가져오는 것을 뜻합니다.

description
이 읽기 전용 어트리뷰트는 마지막 질의의 열 이름을 제공합니다. 파이썬 DB API와의 호환성을 유지하기 위해, 각 열마다 7-튜플을 반환하는데, 각 튜플의 마지막 6개 항목은 *None* 입니다.

일치하는 행이 없는 SELECT 문에도 설정됩니다.

connection
이 읽기 전용 어트리뷰트는 *Cursor* 객체가 사용하는 SQLite 데이터베이스 *Connection*을 제공합니다. *con.cursor()*를 호출하여 생성된 *Cursor* 객체는 *con*을 참조하는 *connection* 어트리뷰트를 가집니다:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4 Row 객체

class sqlite3.**Row**
Row 인스턴스는 *Connection* 객체에 대해 고도로 최적화된 *row_factory* 역할을 합니다. 대부분 기능에서 튜플을 모방하려고 합니다.

열 이름과 인덱스에 의한 매핑 액세스와, 이터레이션, 표현(repr), 동등성 검사 및 *len()*을 지원합니다.

두 개의 *Row* 객체가 정확히 같은 열을 갖고 그 구성원이 같으면 같다고 비교됩니다.

keys()
이 메서드는 열 이름 리스트를 반환합니다. 질의 직후, *Cursor.description*에 있는 각 튜플의 첫 번째 멤버입니다.

버전 3.5에서 변경: 슬라이싱 지원이 추가되었습니다.

위에서 주어진 예제에서처럼 테이블을 초기화한다고 가정해 봅시다:

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

이제 우리는 *Row*를 연결합니다:

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

12.6.5 예외

exception sqlite3.**Warning**
*Exception*의 서브 클래스.

exception sqlite3.**Error**
이 모듈에 있는 다른 예외의 베이스 클래스. *Exception*의 서브 클래스입니다.

exception sqlite3.**DatabaseError**
데이터베이스와 관련된 에러에 대해 발생하는 예외.

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.**ProgrammingError**
프로그래밍 에러에 대한 예외, 예를 들어, 테이블을 찾을 수 없거나 이미 존재 함, SQL 문의 문법 에러, 지정된 매개 변수 개수가 잘못됨 등. *DatabaseError*의 서브 클래스입니다.

exception sqlite3.**OperationalError**
데이터베이스 연산과 관련되고 프로그래머의 제어하에 있지 않은 에러에 관한 오류. 예를 들어, 예기치 않은 단절이 발생하거나, 데이터 소스 이름을 찾을 수 없거나, 트랜잭션이 진행될 수 없을 때 등. *DatabaseError*의 서브 클래스입니다.

exception sqlite3.**NotSupportedError**
데이터베이스에서 지원하지 않는 메서드나 데이터베이스 API가 사용될 때 발생하는 예외. 예를 들어, 트랜잭션을 지원하지 않는 연결에서 *rollback()* 메서드를 호출할 때. *DatabaseError*의 서브 클래스입니다.

12.6.6 SQLite 와 파이썬 형

소개

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*

sqlite3 모듈의 형 시스템은 두 가지 방식으로 확장 가능합니다: 객체 어댑터를 통해 SQLite 데이터베이스에 추가 파이썬 형을 저장할 수 있으며 변환기를 통해 *sqlite3* 모듈에서 SQLite 형을 다른 파이썬 형으로 변환할 수 있습니다.

어댑터를 사용하여 SQLite 데이터베이스에 추가 파이썬 형을 저장하기

앞에서 설명한 것처럼, SQLite는 기본적으로 제한된 형 집합만 지원합니다. SQLite에 다른 파이썬 형을 사용하려면, SQLite에 대해 sqlite3 모듈이 지원하는 형 중 하나로 **어댑트** 해야 합니다: NoneType, int, float, str, bytes 중 하나.

sqlite3 모듈이 사용자 정의 파이썬 형을, 지원되는 형 중 하나로 어댑트하도록 만드는 두 가지 방법이 있습니다.

객체가 스스로 어댑트하도록 하기

여러분이 스스로 클래스를 작성한다면 이것이 좋은 접근법입니다. 다음과 같은 클래스가 있다고 가정해 봅시다:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

이제 Point를 단일 SQLite 열에 저장하려고 합니다. 먼저 포인트를 나타내는데 사용할 지원되는 형 중 하나를 선택해야 합니다. str을 사용하고 좌표를 세미콜론으로 분리하기로 합시다. 그런 다음 여러분의 클래스에 변환된 값을 반환하는 `__conform__(self, protocol)` 메서드를 제공해야 합니다. 매개 변수 *protocol*은 `PrepareProtocol`이 됩니다.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

어댑터 콜러블 등록하기

또 다른 가능성은 형을 문자열 표현으로 변환하는 함수를 만들고. 그 함수를 *register_adapter()*로 등록하는 것입니다.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

sqlite3 모듈에는 파이썬의 내장 *datetime.date* 와 *datetime.datetime* 형에 대한 두 개의 기본 어댑터가 있습니다. 이제 *datetime.datetime* 객체를 ISO 표현이 아닌 유닉스 타임스탬프로 저장하려고 한다고 가정해 봅시다.

```
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])

con.close()
```

SQLite 값을 사용자 정의 파이썬 형으로 변환하기

어댑터를 작성하면 사용자 정의 파이썬 형을 SQLite로 보낼 수 있습니다. 그러나 실제로 유용하게 사용하려면 파이썬에서 SQLite를 거쳐 다시 파이썬으로 돌아오는 순환이 동작하게 할 필요가 있습니다.

변환기를 사용하십시오.

`Point` 클래스로 돌아갑시다. 세미콜론으로 분리된 x와 y 좌표를 SQLite에 문자열로 저장했습니다.

먼저, 문자열을 매개 변수로 받아들이고 이것으로부터 `Point` 객체를 만드는 변환기 함수를 정의합니다.

참고: 변환기 함수는 **항상** SQLite로 보낸 값의 데이터형에 상관없이 *bytes* 객체로 호출됩니다.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

이제 *sqlite3* 모듈이 데이터베이스에서 select 한 것이 실제로 Point임을 알게 해야 합니다. 이렇게 하는 두 가지 방법이 있습니다:

- 선언된 형을 통해 묵시적으로
- 열 이름을 통해 명시적으로

두 가지 방법은 섹션 모듈 함수와 상수 의 상수 *PARSE_DECLTYPES* 와 *PARSE_COLNAMES*에 대한 항목에서 설명합니다.

다음 예는 두 가지 접근법을 보여줍니다.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return ("%f;%f" % (point.x, point.y)).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.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()

######################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.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()
```

기본 어댑터와 변환기

datetime 모듈의 date와 datetime 형에 대한 기본 어댑터가 있습니다. 이것들은 ISO 날짜/ISO 타임스탬프로 SQLite로 보내집니다.

기본 변환기는 *datetime.date*는 "date"라는 이름으로, *datetime.datetime*은 "timestamp"라는 이름으로 등록됩니다.

이런 방법으로, 대부분 추가 작업 없이 파이썬의 날짜/타임스탬프를 사용할 수 있습니다. 어댑터의 형식은 실험적인 SQLite 날짜/시간 함수와도 호환됩니다.

다음 예제는 이를 보여줍니다.

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_
↪COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
↪')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))

con.close()
```

SQLite에 저장된 타임스탬프가 6자리보다 긴 소수부가 있으면, 그 값은 타임스탬프 변환기에 의해 마이크로초 정밀도로 잘립니다.

12.6.7 트랜잭션 제어

하부 sqlite3 라이브러리는 기본적으로 autocommit 모드로 작동하지만, 파이썬 *sqlite3* 모듈은 기본적으로 그렇지 않습니다.

autocommit 모드는 데이터베이스를 수정하는 명령문이 즉시 적용됨을 뜻합니다. BEGIN 이나 SAVEPOINT 문은 autocommit 모드를 비활성화하고, 가장 바깥쪽 트랜잭션을 끝내는 COMMIT, ROLLBACK 또는 RELEASE는 autocommit 모드를 다시 켭니다.

기본적으로 파이썬 *sqlite3* 모듈은 데이터 조작 언어(DML - Data Modification Language) 문 (즉, INSERT/UPDATE/DELETE/REPLACE) 앞에 암묵적으로 BEGIN 문을 넣습니다.

connect() 호출의 *isolation_level* 매개 변수를 통해, 또는 연결의 isolation_level 프로퍼티를 통해, *sqlite3*가 묵시적으로 실행하는 BEGIN 문의 종류를 제어할 수 있습니다. *isolation_level*을 지정하지 않으면, 단순한 BEGIN이 사용되며, 이는 DEFERRED를 지정하는 것과 같습니다. 가능한 다른 값은 IMMEDIATE 와 EXCLUSIVE입니다.

isolation_level를 None로 설정하여 *sqlite3* 모듈의 묵시적 트랜잭션 관리를 비활성화할 수 있습니다. 그러면 하부 sqlite3 라이브러리가 autocommit 모드로 작동합니다. 그런 다음 코드에서 BEGIN, ROLLBACK, SAVEPOINT 및 RELEASE 문을 명시적으로 실행하여 트랜잭션 상태를 완전히 제어할 수 있습니다.

버전 3.6에서 변경: *sqlite3*는 DDL 문 앞에서 열린 트랜잭션을 묵시적으로 커밋했습니다. 더는 그렇지 않습니다.

12.6.8 효율적으로 sqlite3 사용하기

바로 가기 메서드 사용하기

Connection 객체의 비표준 execute(), executemany() 및 executescript() 메서드를 사용하면, (종종 불필요한) *Cursor* 객체를 명시적으로 만들 필요가 없으므로, 코드를 더 간결하게 작성할 수 있습니다. 대신, *Cursor* 객체가 묵시적으로 만들어지며 이러한 바로 가기 메서드는 커서 객체를 반환합니다. 이런 방법으로, *Connection* 객체에 대한 단일 호출만 사용하여 SELECT 문을 실행하고 직접 이터레이트할 수 있습니다.

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print(row)

print("I just deleted", con.execute("delete from person").rowcount, "rows")

# close is not a shortcut method and it's not called automatically,
# so the connection object should be closed manually
con.close()
```

인덱스 대신 이름으로 열 액세스하기

sqlite3 모듈의 유용한 기능 중 하나는 행 팩토리로 사용하도록 설계된 내장 *sqlite3.Row* 클래스입니다.

이 클래스로 감싼 행은 인덱스(튜플처럼)와 대소 문자를 구분하지 않는 이름으로 액세스할 수 있습니다:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]

con.close()
```

컨텍스트 관리자로 연결 사용하기

연결 객체는 트랜잭션을 자동으로 커밋하거나 롤백하는 컨텍스트 관리자로 사용할 수 있습니다. 예외가 발생하면, 트랜잭션이 롤백 됩니다; 그렇지 않으면 트랜잭션이 커밋 됩니다:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# 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 person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

12.6.9 일반적인 문제

다중 스레드

이전 SQLite 버전에서는 스레드 간에 연결을 공유하는 데 문제가 있었습니다. 이것이 파이썬 모듈이 스레드 간에 연결과 커서를 공유하도록 허용하지 않는 이유입니다. 여전히 그렇게 하려고 하면 실행 시간에 예외가 발생합니다.

유일한 예외는 *interrupt()* 메서드를 호출하는 것입니다. 이 메서드는 다른 스레드에서 호출해야만 합니다.

CHAPTER 13

데이터 압축 및 보관

이 장에서 설명하는 모듈은 zlib, gzip, bzip2 및 lzma 알고리즘을 사용한 데이터 압축과 ZIP- 및 tar- 형식 저장소 생성을 지원합니다. `shutil` 모듈에서 제공하는 *Archiving operations*도 참조하십시오.

13.1 `zlib` — Compression compatible with `gzip`

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at http://www.zlib.net. There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib's functions have many options and often need to be used in a particular order. This documentation doesn't attempt to cover all of the permutations; consult the zlib manual at http://www.zlib.net/manual.html for authoritative information.

For reading and writing `.gz` files see the `gzip` module.

The available exception and functions in this module are:

exception `zlib.`**error**
Exception raised on compression and decompression errors.

`zlib.`**adler32**(*data*[, *value*])
Computes an Adler-32 checksum of *data*. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 1 is used. Passing in *value* allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

버전 3.0에서 변경: Always returns an unsigned value. To generate the same numeric value across all Python versions and platforms, use `adler32(data) & 0xffffffff`.

zlib.**compress**(*data*, *level=-1*)

Compresses the bytes in *data*, returning a bytes object containing compressed data. *level* is an integer from 0 to 9 or −1 controlling the level of compression; 1 (Z_BEST_SPEED) is fastest and produces the least compression, 9 (Z_BEST_COMPRESSION) is slowest and produces the most. 0 (Z_NO_COMPRESSION) is no compression. The default value is −1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6). Raises the *error* exception if any error occurs.

버전 3.6에서 변경: *level* can now be used as a keyword parameter.

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

Returns a compression object, to be used for compressing data streams that won't fit into memory at once.

level is the compression level – an integer from 0 to 9 or −1. A value of 1 (Z_BEST_SPEED) is fastest and produces the least compression, while a value of 9 (Z_BEST_COMPRESSION) is slowest and produces the most. 0 (Z_NO_COMPRESSION) is no compression. The default value is −1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

method is the compression algorithm. Currently, the only supported value is DEFLATED.

The *wbits* argument controls the size of the history buffer (or the "window size") used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values, defaulting to 15 (MAX_WBITS):

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- −9 to −15: Uses the absolute value of *wbits* as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic **gzip** header and trailing checksum in the output.

The *memLevel* argument controls the amount of memory used for the internal compression state. Valid values range from 1 to 9. Higher values use more memory, but are faster and produce smaller output.

strategy is used to tune the compression algorithm. Possible values are Z_DEFAULT_STRATEGY, Z_FILTERED, Z_HUFFMAN_ONLY, Z_RLE (zlib 1.2.0.1) and Z_FIXED (zlib 1.2.2.2).

zdict is a predefined compression dictionary. This is a sequence of bytes (such as a *bytes* object) containing subsequences that are expected to occur frequently in the data that is to be compressed. Those subsequences that are expected to be most common should come at the end of the dictionary.

버전 3.3에서 변경: Added the *zdict* parameter and keyword argument support.

zlib.**crc32**(*data*[, *value*])

Computes a CRC (Cyclic Redundancy Check) checksum of *data*. The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 0 is used. Passing in *value* allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

버전 3.0에서 변경: Always returns an unsigned value. To generate the same numeric value across all Python versions and platforms, use `crc32(data) & 0xffffffff`.

zlib.**decompress**(*data*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

Decompresses the bytes in *data*, returning a bytes object containing the uncompressed data. The *wbits* parameter depends on the format of *data*, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the *error* exception if any error occurs.

The *wbits* parameter controls the size of the history buffer (or "window size"), and what header and trailer format is expected. It is similar to the parameter for *compressobj()*, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
- −8 to −15: Uses the absolute value of *wbits* as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an *error* exception. The default *wbits* value corresponds to the largest window size and requires a zlib header and trailer to be included.

bufsize is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don't have to get this value exactly right; tuning it will only save a few calls to `malloc()`.

버전 3.6에서 변경: *wbits* and *bufsize* can be used as keyword arguments.

`zlib.`**`decompressobj`**(*wbits=MAX_WBITS*[, *zdict*])

Returns a decompression object, to be used for decompressing data streams that won't fit into memory at once.

The *wbits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format is expected. It has the same meaning as *described for decompress()*.

The *zdict* parameter specifies a predefined compression dictionary. If provided, this must be the same dictionary as was used by the compressor that produced the data that is to be decompressed.

참고: If *zdict* is a mutable object (such as a *bytearray*), you must not modify its contents between the call to *decompressobj()* and the first call to the decompressor's `decompress()` method.

버전 3.3에서 변경: Added the *zdict* parameter.

Compression objects support the following methods:

`Compress.`**`compress`**(*data*)

Compress *data*, returning a bytes object containing compressed data for at least part of the data in *data*. This data should be concatenated to the output produced by any preceding calls to the *compress()* method. Some input may be kept in internal buffers for later processing.

`Compress.`**`flush`**([*mode*])

All pending input is processed, and a bytes object containing the remaining compressed output is returned. *mode* can be selected from the constants `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK` (zlib 1.2.3.4), or `Z_FINISH`, defaulting to `Z_FINISH`. Except `Z_FINISH`, all constants allow compressing further bytestrings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling *flush()* with *mode* set to `Z_FINISH`, the *compress()* method cannot be called again; the only realistic action is to delete the object.

`Compress.`**`copy`**()

Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

Decompression objects support the following methods and attributes:

Decompress.**unused_data**
A bytes object which contains any bytes past the end of the compressed data. That is, this remains b"" until the last byte that contains compression data is available. If the whole bytestring turned out to contain compressed data, this is b"", an empty bytes object.

Decompress.**unconsumed_tail**
A bytes object that contains any data that was not consumed by the last *decompress()* call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent *decompress()* method call in order to get correct output.

Decompress.**eof**
A boolean indicating whether the end of the compressed data stream has been reached.

This makes it possible to distinguish between a properly-formed compressed stream, and an incomplete or truncated one.

버전 3.3에 추가.

Decompress.**decompress**(*data*, *max_length=0*)
Decompress *data*, returning a bytes object containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the *decompress()* method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is non-zero then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute *unconsumed_tail*. This bytestring must be passed to a subsequent call to *decompress()* if decompression is to continue. If *max_length* is zero then the whole input is decompressed, and *unconsumed_tail* is empty.

버전 3.6에서 변경: *max_length* can be used as a keyword argument.

Decompress.**flush**([*length*])
All pending input is processed, and a bytes object containing the remaining uncompressed output is returned. After calling *flush()*, the *decompress()* method cannot be called again; the only realistic action is to delete the object.

The optional parameter *length* sets the initial size of the output buffer.

Decompress.**copy**()
Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

Information about the version of the zlib library in use is available through the following constants:

zlib.**ZLIB_VERSION**
The version string of the zlib library that was used for building the module. This may be different from the zlib library actually used at runtime, which is available as *ZLIB_RUNTIME_VERSION*.

zlib.**ZLIB_RUNTIME_VERSION**
The version string of the zlib library actually loaded by the interpreter.

버전 3.3에 추가.

더 보기:

Module *gzip* Reading and writing **gzip**-format files.

http://www.zlib.net The zlib library home page.

http://www.zlib.net/manual.html The zlib manual explains the semantics and usage of the library's many functions.

13.2 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.

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.

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.

mtime 인자는 압축할 때 스트림의 마지막 수정 시간 필드에 기록되는 선택적 숫자 타임스탬프입니다. 압축 모드에서만 제공해야 합니다. 생략되거나 None이면, 현재 시각이 사용됩니다. 자세한 내용은 *mtime* 어트리뷰트를 참조하십시오.

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. 호출을 만족시키기 위해 압축된 스트림에 대해 최대 한 번의 읽기가 수행됩니다. 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).

버전 3.2에 추가.

mtime
압축을 풀 때, 가장 최근에 읽은 헤더의 마지막 수정 시간 필드의 값을 이 어트리뷰트에서 정수로 읽을 수 있습니다. 헤더를 읽기 전의 초깃값은 None입니다.

모든 **gzip** 압축 스트림에는 이 타임스탬프 필드가 있어야 합니다. **gunzip**과 같은 일부 프로그램은 타임스탬프를 사용합니다. 형식은 *time.time()*의 반환 값과 *os.stat()*에 의해 반환된 객체의 *st_mtime* 어트리뷰트와 같습니다.

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.

gzip.**compress**(*data*, *compresslevel=9*)
*data*를 압축하여, 압축된 데이터가 포함된 *bytes* 객체를 반환합니다. *compresslevel*은 위의 *GzipFile* 생성자와 같은 의미입니다.

버전 3.2에 추가.

gzip.**decompress**(*data*)
*data*의 압축을 풀어서, 압축되지 않은 데이터가 포함된 *bytes* 객체를 반환합니다.

버전 3.2에 추가.

13.2.1 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.

13.3 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.

이 모듈의 모든 클래스는 다중 스레드에서 안전하게 액세스할 수 있습니다.

13.3.1 파일 압축(해제)

bz2.**open**(*filename*, *mode='r'*, *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.

버전 3.3에 추가.

버전 3.4에서 변경: `'x'` (배타적 생성) 모드가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

class bz2.**BZ2File**(*filename*, *mode='r'*, *buffering=None*, *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.

buffering 인자는 무시됩니다. 이것의 사용은 폐지되었습니다.

*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 메서드도 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).

버전 3.3에 추가.

버전 3.1에서 변경: `with` 문에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: `fileno()`, `readable()`, `seekable()`, `writable()`, `read1()` 및 `readinto()` 메서드가 추가되었습니다.

버전 3.3에서 변경: 실제 파일명 대신 파일 객체 인 *filename*에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: 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.

13.3.2 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 스트림 마다 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`.

버전 3.3에 추가.

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`.

버전 3.5에 추가.

13.3.3 일괄 압축(해제)

bz2.**compress**(*data*, *compresslevel=9*)
비이트열류 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.

13.3.4 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.
... 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
```

13.4 `lzma` — Compression using the LZMA algorithm

버전 3.3에 추가.

Source code: Lib/lzma.py

This module provides classes and convenience functions for compressing and decompressing data using the LZMA compression algorithm. Also included is a file interface supporting the `.xz` and legacy `.lzma` file formats used by the **xz** utility, as well as raw compressed streams.

The interface provided by this module is very similar to that of the *bz2* module. However, note that *LZMAFile* is *not* thread-safe, unlike *bz2.BZ2File*, so if you need to use a single *LZMAFile* instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.**LZMAError**
This exception is raised when an error occurs during compression or decompression, or while initializing the compressor/decompressor state.

13.4.1 Reading and writing compressed files

lzma.**open**(*filename*, *mode="rb"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)
Open an LZMA-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be either an actual 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 any of `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` or `"ab"` for binary mode, or `"rt"`, `"wt"`, `"xt"`, or `"at"` for text mode. The default is `"rb"`.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for *LZMADecompressor*. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for *LZMACompressor*.

For binary mode, this function is equivalent to the *LZMAFile* constructor: `LZMAFile(filename, mode, ...)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a *LZMAFile* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line ending(s).

버전 3.4에서 변경: Added support for the `"x"`, `"xb"` and `"xt"` modes.

버전 3.6에서 변경: Accepts a *path-like object*.

class lzma.**LZMAFile**(*filename=None*, *mode="r"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)
Open an LZMA-compressed file in binary mode.

An *LZMAFile* can wrap an already-open *file object*, or operate directly on a named file. The *filename* argument specifies either the file object to wrap, or the name of the file to open (as a *str*, *bytes* or *path-like* object). When wrapping an existing file object, the wrapped file will not be closed when the *LZMAFile* is closed.

The *mode* argument can be either `"r"` for reading (default), `"w"` for overwriting, `"x"` for exclusive creation, or `"a"` for appending. These can equivalently be given as `"rb"`, `"wb"`, `"xb"` and `"ab"` respectively.

If *filename* 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 opening a file for reading, the input file may be the concatenation of multiple separate compressed streams. These are transparently decoded as a single logical stream.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for *LZMADecompressor*. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for *LZMACompressor*.

LZMAFile supports all the members specified by *io.BufferedIOBase*, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method is 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 *LZMAFile*, it may change the position of the underlying file object (e.g. if the *LZMAFile* was constructed by passing a file object for *filename*).

버전 3.4에서 변경: Added support for the "x" and "xb" modes.

버전 3.5에서 변경: The *read()* method now accepts an argument of None.

버전 3.6에서 변경: Accepts a *path-like object*.

13.4.2 Compressing and decompressing data in memory

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=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 *compress()*.

The *format* argument specifies what container format should be used. Possible values are:

- **FORMAT_XZ: The .xz container format.** This is the default format.
- **FORMAT_ALONE: The legacy .lzma container format.** This format is more limited than .xz – it does not support integrity checks or multiple filters.
- **FORMAT_RAW: A raw data stream, not using any container format.** This format specifier does not support integrity checks, and requires that you always specify a custom filter chain (for both compression and decompression). Additionally, data compressed in this manner cannot be decompressed using FORMAT_AUTO (see *LZMADecompressor*).

The *check* argument specifies the type of integrity check to include in the compressed data. This check is used when decompressing, to ensure that the data has not been corrupted. Possible values are:

- CHECK_NONE: No integrity check. This is the default (and the only acceptable value) for FORMAT_ALONE and FORMAT_RAW.
- CHECK_CRC32: 32-bit Cyclic Redundancy Check.
- CHECK_CRC64: 64-bit Cyclic Redundancy Check. This is the default for FORMAT_XZ.
- CHECK_SHA256: 256-bit Secure Hash Algorithm.

If the specified check is not supported, an *LZMAError* is raised.

The compression settings can be specified either as a preset compression level (with the *preset* argument), or in detail as a custom filter chain (with the *filters* argument).

The *preset* argument (if provided) should be an integer between 0 and 9 (inclusive), optionally OR-ed with the constant PRESET_EXTREME. If neither *preset* nor *filters* are given, the default behavior is to use PRESET_DEFAULT (preset level 6). Higher presets produce smaller output, but make the compression process slower.

참고: In addition to being more CPU-intensive, compression with higher presets also requires much more memory (and produces output that needs more memory to decompress). With preset 9 for example, the overhead for an *LZMACompressor* object can be as high as 800 MiB. For this reason, it is generally best to stick with the default preset.

The *filters* argument (if provided) should be a filter chain specifier. See *Specifying custom filter chains* for details.

compress(*data*)
Compress *data* (a `bytes` object), returning a `bytes` object containing compressed data for at least part of the input. 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()`.

flush()
Finish the compression process, returning a `bytes` object containing any data stored in the compressor's internal buffers.

The compressor cannot be used after this method has been called.

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=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 `decompress()`.

The *format* argument specifies the container format that should be used. The default is `FORMAT_AUTO`, which can decompress both `.xz` and `.lzma` files. Other possible values are `FORMAT_XZ`, `FORMAT_ALONE`, and `FORMAT_RAW`.

The *memlimit* argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an `LZMAError` if it is not possible to decompress the input within the given memory limit.

The *filters* argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if *format* is `FORMAT_RAW`, but should not be used for other formats. See *Specifying custom filter chains* for more information about filter chains.

참고: This class does not transparently handle inputs containing multiple compressed streams, unlike `decompress()` and `LZMAFile`. To decompress a multi-stream input with `LZMADecompressor`, you must create a new decompressor for each stream.

---

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 nonnegative, 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 stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the `unused_data` attribute.

버전 3.5에서 변경: Added the *max_length* parameter.

check
The ID of the integrity check used by the input stream. This may be `CHECK_UNKNOWN` until enough of the input has been decoded to determine what integrity check it uses.

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 uncompressed input.

버전 3.5에 추가.

`lzma.`**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)
Compress *data* (a *bytes* object), returning the compressed data as a *bytes* object.

See *LZMACompressor* above for a description of the *format*, *check*, *preset* and *filters* arguments.

`lzma.`**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
Decompress *data* (a *bytes* object), returning the uncompressed data as a *bytes* object.

If *data* is the concatenation of multiple distinct compressed streams, decompress all of these streams, and return the concatenation of the results.

See *LZMADecompressor* above for a description of the *format*, *memlimit* and *filters* arguments.

13.4.3 Miscellaneous

`lzma.`**is_check_supported**(*check*)
Return `True` if the given integrity check is supported on this system.

`CHECK_NONE` and `CHECK_CRC32` are always supported. `CHECK_CRC64` and `CHECK_SHA256` may be unavailable if you are using a version of **liblzma** that was compiled with a limited feature set.

13.4.4 Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key `"id"`, and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- **Compression filters:**
  - `FILTER_LZMA1` (for use with `FORMAT_ALONE`)
  - `FILTER_LZMA2` (for use with `FORMAT_XZ` and `FORMAT_RAW`)
- **Delta filter:**
  - `FILTER_DELTA`
- **Branch-Call-Jump (BCJ) filters:**
  - `FILTER_X86`
  - `FILTER_IA64`
  - `FILTER_ARM`
  - `FILTER_ARMTHUMB`
  - `FILTER_POWERPC`
  - `FILTER_SPARC`

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- `preset`: A compression preset to use as a source of default values for options that are not specified explicitly.
- `dict_size`: Dictionary size in bytes. This should be between 4 KiB and 1.5 GiB (inclusive).
- `lc`: Number of literal context bits.
- `lp`: Number of literal position bits. The sum `lc + lp` must be at most 4.
- `pb`: Number of position bits; must be at most 4.
- `mode`: `MODE_FAST` or `MODE_NORMAL`.
- `nice_len`: What should be considered a "nice length" for a match. This should be 273 or less.
- `mf`: What match finder to use – `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, or `MF_BT4`.
- `depth`: Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It supports one option, `dist`. This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, `start_offset`. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

13.4.5 Examples

Reading in a compressed file:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creating a compressed file:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Incremental compression:

```
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()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
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")
```

Creating a compressed file using a custom filter chain:

```
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")
```

13.5 zipfile — Work with ZIP archives

Source code: Lib/zipfile.py

The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GiB in size). It supports decryption of encrypted files in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

The module defines the following items:

exception zipfile.**BadZipFile**
The error raised for bad ZIP files.

버전 3.2에 추가.

exception zipfile.**BadZipfile**
Alias of *BadZipFile*, for compatibility with older Python versions.

버전 3.2부터 폐지.

exception zipfile.**LargeZipFile**
The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

class zipfile.**ZipFile**
The class for reading and writing ZIP files. See section *ZipFile Objects* for constructor details.

class zipfile.**PyZipFile**
Class for creating ZIP archives containing Python libraries.

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)
Class used to represent information about a member of an archive. Instances of this class are returned by the *getinfo()* and *infolist()* methods of *ZipFile* objects. Most users of the *zipfile* module will not need to create these, but only use those created by this module. *filename* should be the full name of the archive

member, and *date_time* should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section *ZipInfo Objects*.

zipfile.**is_zipfile**(*filename*)
Returns `True` if *filename* is a valid ZIP file based on its magic number, otherwise returns `False`. *filename* may be a file or file-like object too.

버전 3.1에서 변경: Support for file and file-like objects.

zipfile.**ZIP_STORED**
The numeric constant for an uncompressed archive member.

zipfile.**ZIP_DEFLATED**
The numeric constant for the usual ZIP compression method. This requires the *zlib* module.

zipfile.**ZIP_BZIP2**
The numeric constant for the BZIP2 compression method. This requires the *bz2* module.

버전 3.3에 추가.

zipfile.**ZIP_LZMA**
The numeric constant for the LZMA compression method. This requires the *lzma* module.

버전 3.3에 추가.

참고: The ZIP file format specification has included support for bzip2 compression since 2001, and for LZMA compression since 2006. 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 Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page Information about the Info-ZIP project's ZIP archive programs and development libraries.

13.5.1 ZipFile Objects

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*)
Open a ZIP file, where *file* can be a path to a file (a string), a file-like object or a *path-like object*.

The *mode* parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, `'a'` to append to an existing file, or `'x'` to exclusively create and write a new file. If *mode* is `'x'` and *file* refers to an existing file, a *FileExistsError* will be raised. If *mode* is `'a'` and *file* refers to an existing ZIP file, then additional files are added to it. If *file* does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`). If *mode* is `'a'` and the file does not exist at all, it is created. If *mode* is `'r'` or `'a'`, the file should be seekable.

compression is the ZIP compression method to use when writing the archive, and should be *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA*; unrecognized values will cause *NotImplementedError* to be raised. If *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* is specified but the corresponding module (*zlib*, *bz2* or *lzma*) is not available, *RuntimeError* is raised. The default is *ZIP_STORED*.

If *allowZip64* is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 4 GiB. If it is `false` *zipfile* will raise an exception when the ZIP file would require ZIP64 extensions.

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).

If the file is created with mode `'w'`, `'x'` or `'a'` and then *closed* without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the `with` statement. In the example, *myzip* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

버전 3.2에 추가: Added the ability to use *ZipFile* as a context manager.

버전 3.3에서 변경: Added support for *bzip2* and *lzma* compression.

버전 3.4에서 변경: ZIP64 extensions are enabled by default.

버전 3.5에서 변경: Added support for writing to unseekable streams. Added support for the `'x'` mode.

버전 3.6에서 변경: Previously, a plain *RuntimeError* was raised for unrecognized compression values.

버전 3.6.2에서 변경: The *file* parameter accepts a *path-like object*.

버전 3.7에서 변경: Add the *compresslevel* parameter.

`ZipFile.`**`close`**`()`
Close the archive file. You must call *close()* before exiting your program or essential records will not be written.

`ZipFile.`**`getinfo`**`(`*name*`)`
Return a *ZipInfo* object with information about the archive member *name*. Calling *getinfo()* for a name not currently contained in the archive will raise a *KeyError*.

`ZipFile.`**`infolist`**`()`
Return a list containing a *ZipInfo* object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

`ZipFile.`**`namelist`**`()`
Return a list of archive members by name.

`ZipFile.`**`open`**`(`*name*, *mode='r'*, *pwd=None*, *, *force_zip64=False*`)`
Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the archive or a *ZipInfo* object. The *mode* parameter, if included, must be `'r'` (the default) or `'w'`. *pwd* is the password used to decrypt encrypted ZIP files.

open() is also a context manager and therefore supports the `with` statement:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With *mode* `'r'` the file-like object (`ZipExtFile`) is read-only and provides the following methods: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, `__iter__()`, *__next__()*. These objects can operate independently of the ZipFile.

With `mode='w'`, a writable file handle is returned, which supports the *write()* method. While a writable file handle is open, attempting to read or write other files in the ZIP file will raise a *ValueError*.

When writing a file, if the file size is not known in advance but may exceed 2 GiB, pass `force_zip64=True` to ensure that the header format is capable of supporting large files. If the file size is known in advance, construct a *ZipInfo* object with *file_size* set, and use that as the *name* parameter.

참고: The *open()*, *read()* and *extract()* methods can take a filename or a *ZipInfo* object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

버전 3.6에서 변경: Removed support of `mode='U'`. Use *io.TextIOWrapper* for reading compressed text files in *universal newlines* mode.

버전 3.6에서 변경: *open()* can now be used to write files into the archive with the `mode='w'` option.

버전 3.6에서 변경: Calling *open()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

`ZipFile.`**`extract`**(*member*, *path=None*, *pwd=None*)
Extract a member from the archive to the current working directory; *member* must be its full name or a *ZipInfo* object. Its file information is extracted as accurately as possible. *path* specifies a different directory to extract to. *member* can be a filename or a *ZipInfo* object. *pwd* is the password used for encrypted files.

Returns the normalized path created (a directory or new file).

참고: If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: `///foo/bar` becomes `foo/bar` on Unix, and `C:\foo\bar` becomes `foo\bar` on Windows. And all `".."` components in a member filename will be removed, e.g.: `../../foo../../ba..r` becomes `foo../ba..r`. On Windows illegal characters (`:`, `<`, `>`, `|`, `"`, `?`, and `*`) replaced by underscore (_).

버전 3.6에서 변경: Calling *extract()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

버전 3.6.2에서 변경: The *path* parameter accepts a *path-like object*.

`ZipFile.`**`extractall`**(*path=None*, *members=None*, *pwd=None*)
Extract all members from the archive to the current working directory. *path* specifies a different directory to extract to. *members* is optional and must be a subset of the list returned by *namelist()*. *pwd* is the password used for encrypted files.

경고: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`. This module attempts to prevent that. See *extract()* note.

버전 3.6에서 변경: Calling *extractall()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

버전 3.6.2에서 변경: The *path* parameter accepts a *path-like object*.

`ZipFile.`**`printdir`**()
Print a table of contents for the archive to `sys.stdout`.

`ZipFile.`**`setpassword`**(*pwd*)
Set *pwd* as default password to extract encrypted files.

`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 and, if specified, it will override the default password set with *setpassword()*. Calling *read()* on a ZipFile that uses a compression method other than *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* will raise a *NotImplementedError*. An error will also be raised if the corresponding compression module is not available.

버전 3.6에서 변경: Calling *read()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**testzip**()
Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return None.

버전 3.6에서 변경: Calling *testzip()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**write**(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)
Write the file named *filename* to the archive, giving it the archive name *arcname* (by default, this will be the same as *filename*, but without a drive letter and with leading path separators removed). If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry. Similarly, *compresslevel* will override the constructor if given. The archive must be open with mode 'w', 'x' or 'a'.

참고: Archive names should be relative to the archive root, that is, they should not start with a path separator.

참고: If arcname (or filename, if arcname is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

버전 3.6에서 변경: Calling *write()* on a ZipFile created with mode 'r' or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**writestr**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)
Write a file into the archive. The contents is *data*, which may be either a *str* or a *bytes* instance; if it is a *str*, it is encoded as UTF-8 first. *zinfo_or_arcname* is either the file name it will be given in the archive, or a *ZipInfo* instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode 'w', 'x' or 'a'.

If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry, or in the *zinfo_or_arcname* (if that is a *ZipInfo* instance). Similarly, *compresslevel* will override the constructor if given.

참고: When passing a *ZipInfo* instance as the *zinfo_or_arcname* parameter, the compression method used will be that specified in the *compress_type* member of the given *ZipInfo* instance. By default, the *ZipInfo* constructor sets this member to *ZIP_STORED*.

버전 3.2에서 변경: The *compress_type* argument.

버전 3.6에서 변경: Calling *writestr()* on a ZipFile created with mode 'r' or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

The following data attributes are also available:

ZipFile.**filename**
Name of the ZIP file.

ZipFile.**debug**
The level of debug output to use. This may be set from 0 (the default, no output) to 3 (the most output). Debugging information is written to sys.stdout.

ZipFile.**comment**
The comment associated with the ZIP file as a *bytes* object. If assigning a comment to a *ZipFile* instance

created with mode `'w'`, `'x'` or `'a'`, it should be no longer than 65535 bytes. Comments longer than this will be truncated.

13.5.2 PyZipFile Objects

The *PyZipFile* constructor takes the same parameters as the *ZipFile* constructor, and one additional parameter, *optimize*.

class zipfile.**PyZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)

버전 3.2에 추가: The *optimize* parameter.

버전 3.4에서 변경: ZIP64 extensions are enabled by default.

Instances have one method in addition to those of *ZipFile* objects:

writepy(*pathname*, *basename=''*, *filterfunc=None*)

Search for files `*.py` and add the corresponding file to the archive.

If the *optimize* parameter to *PyZipFile* was not given or `-1`, the corresponding file is a `*.pyc` file, compiling if necessary.

If the *optimize* parameter to *PyZipFile* was `0`, `1` or `2`, only files with that optimization level (see *compile()*) are added to the archive, compiling if necessary.

If *pathname* is a file, the filename must end with `.py`, and just the (corresponding `*.pyc`) file is added at the top level (no path information). If *pathname* is a file that does not end with `.py`, a *RuntimeError* will be raised. If it is a directory, and the directory is not a package directory, then all the files `*.pyc` are added at the top level. If the directory is a package directory, then all `*.pyc` are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively in sorted order.

basename is intended for internal use only.

filterfunc, if given, must be a function taking a single string argument. It will be passed each path (including each individual full file path) before it is added to the archive. If *filterfunc* returns a false value, the path will not be added, and if it is a directory its contents will be ignored. For example, if our test files are all either in `test` directories or start with the string `test_`, we can use a *filterfunc* to exclude them:

```
>>> 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)
```

The *writepy()* method makes archives with file names like this:

```
string.pyc                      # Top level name
test/__init__.pyc               # Package directory
test/testall.pyc                # Module test.testall
test/bogus/__init__.pyc         # Subpackage directory
test/bogus/myfile.pyc           # Submodule test.bogus.myfile
```

버전 3.4에 추가: The *filterfunc* parameter.

버전 3.6.2에서 변경: The *pathname* parameter accepts a *path-like object*.

버전 3.7에서 변경: Recursion sorts directory entries.

13.5.3 ZipInfo Objects

Instances of the *ZipInfo* class are returned by the *getinfo()* and *infolist()* methods of *ZipFile* objects. Each object stores information about a single member of the ZIP archive.

There is one classmethod to make a *ZipInfo* instance for a filesystem file:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*)
Construct a *ZipInfo* instance for a file on the filesystem, in preparation for adding it to a zip file.

filename should be the path to a file or directory on the filesystem.

If *arcname* is specified, it is used as the name within the archive. If *arcname* is not specified, the name will be the same as *filename*, but with any drive letter and leading path separators removed.

버전 3.6에 추가.

버전 3.6.2에서 변경: The *filename* parameter accepts a *path-like object*.

Instances have the following methods and attributes:

ZipInfo.**is_dir**()
Return `True` if this archive member is a directory.

This uses the entry's name: directories should always end with `/`.

버전 3.6에 추가.

ZipInfo.**filename**
Name of the file in the archive.

ZipInfo.**date_time**
The time and date of the last modification to the archive member. This is a tuple of six values:

Index	Value
0	Year (>= 1980)
1	Month (one-based)
2	Day of month (one-based)
3	Hours (zero-based)
4	Minutes (zero-based)
5	Seconds (zero-based)

참고: The ZIP file format does not support timestamps before 1980.

ZipInfo.**compress_type**
Type of compression for the archive member.

ZipInfo.**comment**
Comment for the individual archive member as a *bytes* object.

ZipInfo.**extra**
Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this *bytes* object.

ZipInfo.**create_system**
System which created ZIP archive.

ZipInfo.**create_version**
PKZIP version which created ZIP archive.

ZipInfo.**extract_version**
PKZIP version needed to extract archive.

ZipInfo.**reserved**
Must be zero.

ZipInfo.**flag_bits**
ZIP flag bits.

ZipInfo.**volume**
Volume number of file header.

ZipInfo.**internal_attr**
Internal attributes.

ZipInfo.**external_attr**
External file attributes.

ZipInfo.**header_offset**
Byte offset to the file header.

ZipInfo.**CRC**
CRC-32 of the uncompressed file.

ZipInfo.**compress_size**
Size of the compressed data.

ZipInfo.**file_size**
Size of the uncompressed file.

13.5.4 Command-Line Interface

The *zipfile* module provides a simple command-line interface to interact with ZIP archives.

If you want to create a new ZIP archive, specify its name after the *-c* option and then list the filename(s) that should be included:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

If you want to extract a ZIP archive into the specified directory, use the *-e* option:

```
$ python -m zipfile -e monty.zip target-dir/
```

For a list of the files in a ZIP archive, use the *-l* option:

```
$ python -m zipfile -l monty.zip
```

Command-line options

-l <zipfile>
--list <zipfile>
List files in a zipfile.

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
Create zipfile from source files.

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
Extract zipfile into target directory.

-t <zipfile>
--test <zipfile>
Test whether the zipfile is valid or not.

13.6 tarfile — Read and write tar archive files

Source code: Lib/tarfile.py

The *tarfile* module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the *zipfile* module to read or write `.zip` files, or the higher-level functions in *shutil*.

Some facts and figures:

- reads and writes *gzip*, *bz2* and *lzma* compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including *longname* and *longlink* extensions, read-only support for all variants of the *sparse* extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

버전 3.3에서 변경: Added support for *lzma* compression.

tarfile.**open**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)
Return a *TarFile* object for the pathname *name*. For detailed information on *TarFile* objects and the keyword arguments that are allowed, see *TarFile Objects*.

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

mode	action
`'r'` or `'r:*'`	Open for reading with transparent compression (recommended).
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	Open for reading with gzip compression.
`'r:bz2'`	Open for reading with bzip2 compression.
`'r:xz'`	Open for reading with lzma compression.
`'x'` or `'x:'`	Create a tarfile exclusively without compression. Raise an *FileExistsError* exception if it already exists.
`'x:gz'`	Create a tarfile with gzip compression. Raise an *FileExistsError* exception if it already exists.
`'x:bz2'`	Create a tarfile with bzip2 compression. Raise an *FileExistsError* exception if it already exists.
`'x:xz'`	Create a tarfile with lzma compression. Raise an *FileExistsError* exception if it already exists.
`'a'` or `'a:'`	Open for appending with no compression. The file is created if it does not exist.
`'w'` or `'w:'`	Open for uncompressed writing.
`'w:gz'`	Open for gzip compressed writing.
`'w:bz2'`	Open for bzip2 compressed writing.
`'w:xz'`	Open for lzma compressed writing.

Note that `'a:gz'`, `'a:bz2'` or `'a:xz'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, *ReadError* is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, *CompressionError* is raised.

If *fileobj* is specified, it is used as an alternative to a *file object* opened in binary mode for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, `'x:gz'`, `'x:bz2'`, *tarfile.open()* accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

For special purposes, there is a second format for *mode*: `'filemode|[compression]'`. *tarfile.open()* will return a *TarFile* object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a `read()` or `write()` method (depending on the *mode*). *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin`, a socket *file object* or a tape device. However, such a *TarFile* object is limited in that it does not allow random access, see *Examples*. The currently possible modes:

Mode	Action
`'r|*'`	Open a *stream* of tar blocks for reading with transparent compression.
`'r|'`	Open a *stream* of uncompressed tar blocks for reading.
`'r|gz'`	Open a gzip compressed *stream* for reading.
`'r|bz2'`	Open a bzip2 compressed *stream* for reading.
`'r|xz'`	Open an lzma compressed *stream* for reading.
`'w|'`	Open an uncompressed *stream* for writing.
`'w|gz'`	Open a gzip compressed *stream* for writing.
`'w|bz2'`	Open a bzip2 compressed *stream* for writing.
`'w|xz'`	Open an lzma compressed *stream* for writing.

버전 3.5에서 변경: The `'x'` (exclusive creation) mode was added.

버전 3.6에서 변경: The *name* parameter accepts a *path-like object*.

class tarfile.**TarFile**
Class for reading and writing tar archives. Do not use this class directly: use *tarfile.open()* instead. See *TarFile Objects*.

tarfile.**is_tarfile**(*name*)
Return *True* if *name* is a tar archive file, that the *tarfile* module can read.

The *tarfile* module defines the following exceptions:

exception tarfile.**TarError**
Base class for all *tarfile* exceptions.

exception tarfile.**ReadError**
Is raised when a tar archive is opened, that either cannot be handled by the *tarfile* module or is somehow invalid.

exception tarfile.**CompressionError**
Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.**StreamError**
Is raised for the limitations that are typical for stream-like *TarFile* objects.

exception tarfile.**ExtractError**
Is raised for *non-fatal* errors when using *TarFile.extract()*, but only if `TarFile.errorlevel== 2`.

exception tarfile.**HeaderError**
Is raised by *TarInfo.frombuf()* if the buffer it gets is invalid.

The following constants are available at the module level:

tarfile.**ENCODING**
The default character encoding: `'utf-8'` on Windows, the value returned by *sys.getfilesystemencoding()* otherwise.

Each of the following constants defines a tar archive format that the *tarfile* module is able to create. See section *Supported tar formats* for details.

tarfile.**USTAR_FORMAT**
POSIX.1-1988 (ustar) format.

tarfile.**GNU_FORMAT**
GNU tar format.

tarfile.**PAX_FORMAT**
POSIX.1-2001 (pax) format.

tarfile.**DEFAULT_FORMAT**
The default format for creating archives. This is currently *GNU_FORMAT*.

더 보기:

Module *zipfile* Documentation of the *zipfile* standard module.

Archiving operations Documentation of the higher-level archiving facilities provided by the standard *shutil* module.

GNU tar manual, Basic Tar Format Documentation for tar archive files, including GNU tar extensions.

13.6.1 TarFile Objects

The *TarFile* object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a *TarInfo* object, see *TarInfo Objects* for details.

A *TarFile* object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the *Examples* section for a use case.

버전 3.2에 추가: Added support for the context management protocol.

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*)

All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. *name* may be a *path-like object*. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s mode. *fileobj* will be used from position 0.

참고: *fileobj* is not closed, when *TarFile* is closed.

---

format controls the archive format. It must be one of the constants *USTAR_FORMAT*, *GNU_FORMAT* or *PAX_FORMAT* that are defined at module level.

The *tarinfo* argument can be used to replace the default *TarInfo* class with a different one.

If *dereference* is *False*, add symbolic and hard links to the archive. If it is *True*, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If *ignore_zeros* is *False*, treat an empty block as the end of the archive. If it is *True*, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from `0` (no debug messages) up to `3` (all debug messages). The messages are written to `sys.stderr`.

If *errorlevel* is `0`, all errors are ignored when using *TarFile.extract()*. Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If `1`, all *fatal* errors are raised as *OSError* exceptions. If `2`, all *non-fatal* errors are raised as *TarError* exceptions as well.

The *encoding* and *errors* arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section *Unicode issues* for in-depth information.

The *pax_headers* argument is an optional dictionary of strings which will be added as a pax global header if *format* is *PAX_FORMAT*.

버전 3.2에서 변경: Use `'surrogateescape'` as the default for the *errors* argument.

버전 3.5에서 변경: The `'x'` (exclusive creation) mode was added.

버전 3.6에서 변경: The *name* parameter accepts a *path-like object*.

classmethod TarFile.**open**(...)
Alternative constructor. The *tarfile.open()* function is actually a shortcut to this classmethod.

TarFile.**getmember**(*name*)
Return a *TarInfo* object for member *name*. If *name* can not be found in the archive, *KeyError* is raised.

참고: If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

TarFile.**getmembers**()
Return the members of the archive as a list of *TarInfo* objects. The list has the same order as the members in the archive.

TarFile.**getnames**()
Return the members as a list of their names. It has the same order as the list returned by *getmembers()*.

TarFile.**list**(*verbose=True*, *, *members=None*)
Print a table of contents to `sys.stdout`. If *verbose* is *False*, only the names of the members are printed. If it is *True*, output similar to that of **`ls -l`** is produced. If optional *members* is given, it must be a subset of the list returned by *getmembers()*.

버전 3.5에서 변경: Added the *members* parameter.

TarFile.**next**()
Return the next member of the archive as a *TarInfo* object, when *TarFile* is opened for reading. Return *None* if there is no more available.

TarFile.**extractall**(*path="."*, *members=None*, *, *numeric_owner=False*)
Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by *getmembers()*. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

If *numeric_owner* is *True*, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

경고: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`.

버전 3.5에서 변경: Added the *numeric_owner* parameter.

버전 3.6에서 변경: The *path* parameter accepts a *path-like object*.

TarFile.**extract**(*member*, *path=""*, *set_attrs=True*, *, *numeric_owner=False*)
Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a *TarInfo* object. You can specify a different directory using *path*. *path* may be a *path-like object*. File attributes (owner, mtime, mode) are set unless *set_attrs* is false.

If *numeric_owner* is *True*, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

참고: The *extract()* method does not take care of several extraction issues. In most cases you should consider

using the *extractall()* method.

경고: See the warning for *extractall()*.

버전 3.2에서 변경: Added the *set_attrs* parameter.

버전 3.5에서 변경: Added the *numeric_owner* parameter.

버전 3.6에서 변경: The *path* parameter accepts a *path-like object*.

TarFile.**extractfile**(*member*)
Extract a member from the archive as a file object. *member* may be a filename or a *TarInfo* object. If *member* is a regular file or a link, an *io.BufferedReader* object is returned. Otherwise, *None* is returned.

버전 3.3에서 변경: Return an *io.BufferedReader* object.

TarFile.**add**(*name*, *arcname=None*, *recursive=True*, *, *filter=None*)
Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to *False*. Recursion adds entries in sorted order. If *filter* is given, it should be a function that takes a *TarInfo* object argument and returns the changed *TarInfo* object. If it instead returns *None* the *TarInfo* object will be excluded from the archive. See *Examples* for an example.

버전 3.2에서 변경: Added the *filter* parameter.

버전 3.7에서 변경: Recursion adds entries in sorted order.

TarFile.**addfile**(*tarinfo*, *fileobj=None*)
Add the *TarInfo* object *tarinfo* to the archive. If *fileobj* is given, it should be a *binary file*, and `tarinfo.size` bytes are read from it and added to the archive. You can create *TarInfo* objects directly, or by using *gettarinfo()*.

TarFile.**gettarinfo**(*name=None*, *arcname=None*, *fileobj=None*)
Create a *TarInfo* object from the result of *os.stat()* or equivalent on an existing file. The file is either named by *name*, or specified as a *file object* *fileobj* with a file descriptor. *name* may be a *path-like object*. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*'s *name* attribute, or the *name* argument. The name should be a text string.

You can modify some of the *TarInfo*'s attributes before you add it using *addfile()*. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as *size* may need modifying. This is the case for objects such as *GzipFile*. The *name* may also be modified, in which case *arcname* could be a dummy string.

버전 3.6에서 변경: The *name* parameter accepts a *path-like object*.

TarFile.**close**()
Close the *TarFile*. In write mode, two finishing zero blocks are appended to the archive.

TarFile.**pax_headers**
A dictionary containing key-value pairs of pax global headers.

13.6.2 TarInfo Objects

A *TarInfo* object represents one member in a *TarFile*. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does *not* contain the file's data itself.

TarInfo objects are returned by *TarFile*'s methods `getmember()`, `getmembers()` and `gettarinfo()`.

class tarfile.**TarInfo**(*name=""*)
Create a *TarInfo* object.

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)
Create and return a *TarInfo* object from string buffer *buf*.

Raises *HeaderError* if the buffer is invalid.

classmethod TarInfo.**fromtarfile**(*tarfile*)
Read the next member from the *TarFile* object *tarfile* and return it as a *TarInfo* object.

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)
Create a string buffer from a *TarInfo* object. For information on the arguments see the constructor of the *TarFile* class.

버전 3.2에서 변경: Use `'surrogateescape'` as the default for the *errors* argument.

A `TarInfo` object has the following public data attributes:

TarInfo.**name**
Name of the archive member.

TarInfo.**size**
Size in bytes.

TarInfo.**mtime**
Time of last modification.

TarInfo.**mode**
Permission bits.

TarInfo.**type**
File type. *type* is usually one of these constants: `REGTYPE`, `AREGTYPE`, `LNKTYPE`, `SYMTYPE`, `DIRTYPE`, `FIFOTYPE`, `CONTTYPE`, `CHRTYPE`, `BLKTYPE`, `GNUTYPE_SPARSE`. To determine the type of a *TarInfo* object more conveniently, use the `is*()` methods below.

TarInfo.**linkname**
Name of the target file name, which is only present in *TarInfo* objects of type `LNKTYPE` and `SYMTYPE`.

TarInfo.**uid**
User ID of the user who originally stored this member.

TarInfo.**gid**
Group ID of the user who originally stored this member.

TarInfo.**uname**
User name.

TarInfo.**gname**
Group name.

TarInfo.**pax_headers**
A dictionary containing key-value pairs of an associated pax extended header.

A *TarInfo* object also provides some convenient query methods:

TarInfo.**isfile**()
Return *True* if the Tarinfo object is a regular file.

TarInfo.**isreg**()
Same as *isfile()*.

TarInfo.**isdir**()
Return *True* if it is a directory.

TarInfo.**issym**()
Return *True* if it is a symbolic link.

TarInfo.**islnk**()
Return *True* if it is a hard link.

TarInfo.**ischr**()
Return *True* if it is a character device.

TarInfo.**isblk**()
Return *True* if it is a block device.

TarInfo.**isfifo**()
Return *True* if it is a FIFO.

TarInfo.**isdev**()
Return *True* if it is one of character device, block device or FIFO.

13.6.3 Command-Line Interface

버전 3.4에 추가.

The *tarfile* module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the *-c* option and then list the filename(s) that should be included:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the *-e* option:

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory's name:

```
$ python -m tarfile -e monty.tar  other-dir/
```

For a list of the files in a tar archive, use the *-l* option:

```
$ python -m tarfile -l monty.tar
```

Command-line options

-l <tarfile>
--list <tarfile>
List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
Create tarfile from source files.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
Extract tarfile into the current directory if *output_dir* is not specified.

-t <tarfile>
--test <tarfile>
Test whether the tarfile is valid or not.

-v, --verbose
Verbose output.

13.6.4 Examples

How to extract an entire tar archive to the current working directory:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with *TarFile.extractall()* using a generator function instead of a list:

```
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()
```

How to create an uncompressed tar archive from a list of filenames:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to read a gzip compressed tar archive and display some member information:

```
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()
```

How to create an archive and reset the user information using the *filter* parameter in *TarFile.add()*:

```
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()
```

13.6.5 Supported tar formats

There are three tar formats that can be created with the *tarfile* module:

- The POSIX.1-1988 ustar format (*USTAR_FORMAT*). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.
- The GNU tar format (*GNU_FORMAT*). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. *tarfile* fully supports the GNU tar extensions for long names, sparse file support is read-only.
- The POSIX.1-2001 pax format (*PAX_FORMAT*). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.

  The *pax* format is an extension to the existing *ustar* format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

13.6.6 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-*ASCII* characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding *UTF-8*.

The details of character conversion in *tarfile* are controlled by the *encoding* and *errors* keyword arguments of the *TarFile* class.

encoding defines the character encoding to use for the metadata in the archive. The default value is *sys.getfilesystemencoding()* or `'ascii'` as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If *encoding* is not set appropriately, this conversion may fail.

The *errors* argument defines how characters are treated that cannot be converted. Possible values are listed in section *Error Handlers*. The default scheme is `'surrogateescape'` which Python also uses for its file system calls, see 파일명, 명령 줄 인자 및 환경 변수.

In case of *PAX_FORMAT* archives, *encoding* is generally not needed because all the metadata is stored using *UTF-8*. *encoding* is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.

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 이터레이트 하는 판독기(reader) 객체를 반환합니다. *csvfile*은 이터레이터 프로토콜을 지원하고 `__next__()` 메서드가 호출될 때마다 문자열을 반환하는 객체여야 합니다 — 파일 객체와 리스트 객체 모두 적합합니다. *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.

csv 파일에서 읽은 각 행(row)은 문자열 리스트로 반환됩니다. `QUOTE_NONNUMERIC` 포맷 옵션을 지정하지 않으면 아무런 자동 데이터형 변환도 수행되지 않습니다 (지정하면 인용되지 않은 필드는 float로 변환됩니다).

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.

[1] `newline=''`을 지정하지 않으면, 따옴표 처리된 필드에 포함된 줄 넘김 문자가 올바르게 해석되지 않으며, 줄 끝 표시에 `\r\n`을 사용하는 플랫폼에서 쓸 때 여분의 `\r`이 추가됩니다. csv 모듈은 자체 (유니버설) 줄 넘김 처리를 하므로, `newline=''`을 지정하는 것은 항상 안전합니다.

csv.**unregister_dialect**(*name*)
방언(dialect) 등록소에서 *name*과 관련된 연관된 방언을 삭제합니다. *name*이 등록된 방언 이름이 아니면 *Error*가 발생합니다.

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 *OrderedDict*로 매핑하는 객체를 만듭니다.

fieldnames 매개 변수는 시퀀스입니다. *fieldnames*를 생략하면, 파일 *f*의 첫 번째 행에 있는 값들을 fieldnames로 사용합니다. 필드 이름이 어떻게 결정되는지와 관계없이, 순서 있는 딕셔너리는 원래 순서를 유지합니다.

If a row has more fields than fieldnames, the remaining data is put in a list and stored with the fieldname specified by *restkey* (which defaults to `None`). If a non-blank row has fewer fields than fieldnames, the missing values are filled-in with the value of *restval* (which defaults to `None`).

다른 모든 선택적 또는 키워드 인자는 하부 *reader* 인스턴스에 전달됩니다.

버전 3.6에서 변경: 반환된 행은 이제 `OrderedDict` 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)
OrderedDict([('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.

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**
Dialect 클래스는 어트리뷰트에 주로 의존하는 컨테이너 클래스인데, 특정 *reader*나 *writer* 인스턴스에 대한 파라미터를 정의하는 데 사용됩니다.

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.

버전 3.2에 추가.

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.

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)
    # ... process CSV file contents here ...
```

csv 모듈은 다음 상수를 정의합니다:

csv.**QUOTE_ALL**
writer 객체에 모든 필드를 인용 처리하도록 지시합니다.

csv.**QUOTE_MINIMAL**
writer 객체에 *delimiter*, *quotechar* 또는 *lineterminator*에 들어있는 모든 문자와 같은 특수 문자를 포함하는 필드만 인용 처리하도록 지시합니다.

csv.**QUOTE_NONNUMERIC**
writer 객체에 모든 숫자가 아닌 필드를 인용 처리하도록 지시합니다.

판독기에 인용 처리되지 않은 모든 필드를 *float* 형으로 변환하도록 지시합니다.

csv.**QUOTE_NONE**
writer 객체에 필드를 절대 인용 처리하지 않도록 지시합니다. 출력 데이터에 현재 *delimiter*가 등장하면, 현재 *escapechar* 문자를 앞에 붙입니다. *escapechar*가 설정되지 않았을 때 작성기는 이스케이프 해야 하는 문자가 있으면 *Error*를 발생시킵니다.

*reader*에게 인용 문자의 특별한 처리를 수행하지 않도록 지시합니다.

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 특정 메서드 집합과 단일 `validate()` 메서드가 있는 *Dialect* 클래스의 서브 클래스입니다. *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**
*quoting*이 *QUOTE_NONE*으로 설정되었을 때 *delimiter*를, *doublequote*가 *False*일 때 *quotechar*를 이스케이프 하는데 기록기가 사용하는 한 문자로 된 문자열. 판독 시에, *escapechar*는 뒤따르는 문자에서 특별한 의미를 제거합니다. 기본값은 *None*이며, 이스케이핑을 비활성화합니다.

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**
*delimiter*나 *quotechar*와 같은 특수 문자를 포함하거나 개행 문자를 포함하는 필드를 인용 처리하는 데 사용되는 한 문자라도 된 문자열. 기본값은 `'"'`입니다.

Dialect.**quoting**
언제 인용 기호를 기록기가 생성하고 판독기가 인식해야 하는지를 제어합니다. `QUOTE_*` 상수 (모듈 내용 절을 참조하십시오) 중 하나를 취할 수 있으며 기본값은 *QUOTE_MINIMAL*입니다.

Dialect.**skipinitialspace**
*True*일 때, *delimiter* 바로 뒤에 오는 공백은 무시됩니다. 기본값은 *False*입니다.

Dialect.**strict**
True일 때, 잘못된 CSV 입력에서 예외 *Error*를 발생시킵니다. 기본값은 False입니다.

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 방언에 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:

csvreader.**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 방언에 따라 포매팅해서, 기록기의 파일 객체에 씁니다.

  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 기록합니다.

  버전 3.2에 추가.

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.getpreferredencoding()*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('file {}, line {}: {}'.format(filename, 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 — Configuration file parser

Source code: Lib/configparser.py

This module provides the *ConfigParser* class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

참고: This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

더 보기:

Module *shlex* Support for creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

Module *json* The json module implements a subset of JavaScript syntax which can also be used for this purpose.

14.2.1 Quick Start

Let's take a very basic configuration file that looks like this:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described *in the following section*. Essentially, the file consists of sections, each of which contains keys with values. *configparser* classes can read and write such files. Let's start by creating the above configuration file programmatically.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

As you can see, we can treat a config parser much like a dictionary. There are differences, *outlined later*, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let's read it back and explore the data it holds.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```

As we can see above, the API is pretty straightforward. The only bit of magic involves the `DEFAULT` section which provides default values for all other sections[1]. Note also that keys in sections are case-insensitive and stored in lowercase[1].

[1] Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the *Customizing Parser Behaviour* section.

14.2.2 Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide *`getboolean()`*. This method is case-insensitive and recognizes Boolean values from `'yes'`/`'no'`, `'on'`/`'off'`, `'true'`/`'false'` and `'1'`/`'0'`[1]. For example:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

Apart from *`getboolean()`*, config parsers also provide equivalent *`getint()`* and *`getfloat()`* methods. You can register your own converters and customize the provided ones.[1]

14.2.3 Fallback Values

As with a dictionary, you can use a section's `get()` method to provide fallback values:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the `'CompressionLevel'` key was specified only in the `'DEFAULT'` section. If we try to get it from the section `'topsecret.server.com'`, we will always get the default, even if we specify a fallback:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level `get()` method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument:

```
>>> config.get('bitbucket.org', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

The same `fallback` argument can be used with the *`getint()`*, *`getfloat()`* and *`getboolean()`* methods, for example:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 Supported INI File Structure

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or `:` by default[1]). By default, section names are case sensitive but keys are not[1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted, in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

Configuration files may include comments, prefixed by specific characters (`#` and `;` by default[1]). Comments may appear on their own on an otherwise empty line, possibly indented.[1]

For example:

```
[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
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 Interpolation of values

On top of the core functionality, *ConfigParser* supports interpolation. This means values can be preprocessed before returning them from `get()` calls.

class configparser.**BasicInterpolation**

The default implementation used by *ConfigParser*. It enables values to contain format strings which refer to other values in the same section, or values in the special default section[1]. Additional default values can be provided on initialization.

For example:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
gain: 80%%  # use a %% to escape the % sign (% is the only character that needs
↪to be escaped)
```

In the example above, *ConfigParser* with *interpolation* set to `BasicInterpolation()` would resolve `%(home_dir)s` to the value of `home_dir` (`/Users` in this case). `%(my_dir)s` in effect would resolve to `/Users/lumberjack`. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With `interpolation` set to `None`, the parser would simply return `%(my_dir)s/Pictures` as the value of `my_pictures` and `%(home_dir)s/lumberjack` as the value of `my_dir`.

class configparser.**ExtendedInterpolation**

An alternative handler for interpolation which implements a more advanced syntax, used for instance in `zc.buildout`. Extended interpolation is using `${section:option}` to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the `section:` part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
cost: $$80  # use a $$ to escape the $ sign ($ is the only character that needs
↪to be escaped)
```

Values from other sections can be fetched as well:

```
[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}
```

14.2.6 Mapping Protocol Access

버전 3.2에 추가.

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of *configparser*, the mapping interface implementation is using the `parser['section']['option']` notation.

`parser['section']` in particular returns a proxy for the section's data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What's even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

configparser objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the *MutableMapping* ABC. However, there are a few differences that should be taken into account:

- By default, all keys in sections are accessible in a case-insensitive manner[1]. E.g. `for option in parser["section"]` yields only `optionxform`'ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key `'a'`, both expressions return `True`:

```
"a" in parser["section"]
"A" in parser["section"]
```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a *KeyError*.
- `DEFAULTSECT` cannot be removed from the parser:
  - trying to delete it raises *ValueError*,
  - `parser.clear()` leaves it intact,
  - `parser.popitem()` never returns it.
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic configparser API.
- `parser.items()` is compatible with the mapping protocol (returns a list of *section_name*, *section_proxy* pairs including the DEFAULTSECT). However, this method can also be invoked with arguments: `parser.`

`items(section, raw, vars)`. The latter call returns a list of *option*, *value* pairs for a specified `section`, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

14.2.7 Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. *configparser* goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it's very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the `__init__()` options:

- *defaults*, default value: `None`

  This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don't specify values which are the same as the documented default.

  Hint: if you want to specify default values for a specific section, use `read_dict()` before you read the actual file.

- *dict_type*, default value: *collections.OrderedDict*

  This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the default ordered dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

  An alternative dictionary type can be used for example to sort sections and options on write-back. You can also use a regular dictionary for performance reasons.

  Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys will be ordered because dict preserves order from Python 3.7. For example:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                '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*, default value: `False`

  Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by *configparser*. The *allow_no_value* parameter to the constructor can be used to indicate that such values should be accepted:

```
>>> import configparser
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*, default value: `('=', ':')`

  Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

  See also the *space_around_delimiters* argument to *ConfigParser.write()*.

- *comment_prefixes*, default value: `('#', ';')`

- *inline_comment_prefixes*, default value: `None`

  Comment prefixes are strings that indicate the start of a valid comment within a config file. *comment_prefixes* are used only on otherwise empty lines (optionally indented) whereas *inline_comment_prefixes* can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and `'#'` and `';'` are used as prefixes for whole line comments.

  버전 3.2에서 변경: In previous versions of *configparser* behaviour matched `comment_prefixes=('#',';')` and `inline_comment_prefixes=(';',)`.

  Please note that config parsers don't support escaping of comment prefixes so using *inline_comment_prefixes* may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting *inline_comment_prefixes*. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... 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*, default value: `True`

  When set to `True`, the parser will not allow for any section or option duplicates while reading from a single source (using `read_file()`, `read_string()` or `read_dict()`). It is recommended to use strict parsers in new applications.

  버전 3.2에서 변경: In previous versions of *configparser* behaviour matched `strict=False`.

- *empty_lines_in_values*, default value: `True`

  In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:

```
[Section]
key = multiline
  value with a gotcha

 this = is still a part of the multiline value of 'key'
```

  This can be especially problematic for the user to see if she's using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, `key` and `this`.

- *default_section*, default value: `configparser.DEFAULTSECT` (that is: `"DEFAULT"`)

  The convention of allowing a special section of default values for other sections or interpolation purposes is a

powerful concept of this library, letting users create complex declarative configurations. This section is normally called `"DEFAULT"` but this can be customized to point to any other valid section name. Some typical values include: `"general"` or `"common"`. The name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the `parser_instance.default_section` attribute and may be modified at runtime (i.e. to convert files from one format to another).

- *interpolation*, default value: `configparser.BasicInterpolation`

  Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*. *`RawConfigParser`* has a default value of `None`.

- *converters*, default value: not set

  Config parsers provide option value getters that perform type conversion. By default *`getint()`*, *`getfloat()`*, and *`getboolean()`* are implemented. Should other getters be desirable, users may define them in a subclass or pass a dictionary where each key is a name of the converter and each value is a callable implementing said conversion. For instance, passing `{'decimal': decimal.Decimal}` would add `getdecimal()` on both the parser object and all section proxies. In other words, it will be possible to write both `parser_instance.getdecimal('section', 'key', fallback=0)` and `parser_instance['section'].getdecimal('key', 0)`.

  If the converter needs to access the state of the parser, it can be implemented as a method on a config parser subclass. If the name of this method starts with `get`, it will be available on all section proxies, in the dict-compatible form (see the `getdecimal()` example above).

More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overridden by subclasses or by attribute assignment.

`ConfigParser.`**`BOOLEAN_STATES`**

By default when using *`getboolean()`*, config parsers consider the following values `True`: `'1'`, `'yes'`, `'true'`, `'on'` and the following values `False`: `'0'`, `'no'`, `'false'`, `'off'`. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

```
>>> 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
```

Other typical Boolean pairs include `accept`/`reject` or `enabled`/`disabled`.

`ConfigParser.`**`optionxform`**`(`*`option`*`)`

This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that's unsuitable. For example:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... """
>>> 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: The optionxform function transforms option names to a canonical form. This should be an idempotent function: if the name is already in canonical form, it should be returned unchanged.

ConfigParser.**SECTCRE**

A compiled regular expression used to parse section headers. The default matches `[section]` to the name `"section"`. Whitespace is considered part of the section name, thus `[ larch ]` will be read as a section of name `" larch "`. Override this attribute if that's unsuitable. For example:

```
>>> 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: While ConfigParser objects also use an `OPTCRE` attribute for recognizing option lines, it's not recommended to override it because that would interfere with constructor options *allow_no_value* and *delimiters*.

14.2.8 Legacy API Examples

Mainly because of backwards compatibility concerns, *configparser* provides also a legacy API with explicit `get`/`set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
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!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use *ConfigParser*:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
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."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
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!"
```

14.2.9 ConfigParser Objects

class configparser.**ConfigParser**(*defaults=None*, *dict_type=collections.OrderedDict*, *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={}*)

The main configuration parser. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When *delimiters* is given, it is used as the set of substrings that divide keys from values. When *comment_prefixes* is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When *inline_comment_prefixes* is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When *strict* is `True` (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising *DuplicateSectionError* or *DuplicateOptionError*.

When *empty_lines_in_values* is `False` (default: `True`), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When *allow_no_value* is `True` (default: `False`), options without values are accepted; the value held for these is `None` and they are serialized without the trailing delimiter.

When *default_section* is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named `"DEFAULT"`). This value can be retrieved and changed on runtime using the `default_section` instance attribute.

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*.

All option names used in interpolation will be passed through the *optionxform()* method just like any other option name reference. For example, using the default implementation of *optionxform()* (which converts option names to lower case), the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

When *converters* is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding `get*()` method on the parser object and section proxies.

버전 3.1에서 변경: The default *dict_type* is *collections.OrderedDict*.

버전 3.2에서 변경: *allow_no_value, delimiters, comment_prefixes, strict, empty_lines_in_values, default_section* and *interpolation* were added.

버전 3.5에서 변경: The *converters* argument was added.

버전 3.7에서 변경: The *defaults* argument is read with *read_dict()*, providing consistent behavior across the parser: non-string keys and values are implicitly converted to strings.

defaults()
Return a dictionary containing the instance-wide defaults.

sections()
Return a list of the sections available; the *default section* is not included in the list.

add_section(*section*)
Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised. The name of the section must be a string; if not, *TypeError* is raised.

버전 3.2에서 변경: Non-string section names raise *TypeError*.

has_section(*section*)
Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

options(*section*)
Return a list of options available in the specified *section*.

has_option(*section*, *option*)
If the given *section* exists, and contains the given *option*, return *True*; otherwise return *False*. If the specified *section* is *None* or an empty string, DEFAULT is assumed.

read(*filenames*, *encoding=None*)
Attempt to read and parse an iterable of filenames, returning a list of filenames which were successfully parsed.

If *filenames* is a string, a *bytes* object or a *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the *ConfigParser* instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using *read_file()* before calling *read()* for any optional files:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

버전 3.2에 추가: The *encoding* parameter. Previously, all files were read using the default encoding for *open()*.

버전 3.6.1에 추가: The *filenames* parameter accepts a *path-like object*.

버전 3.7에 추가: The *filenames* parameter accepts a *bytes* object.

read_file(*f*, *source=None*)
Read and parse configuration data from *f* which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument *source* specifies the name of the file being read. If not given and *f* has a `name` attribute, that is used for *source*; the default is `'<???>'`.

버전 3.2에 추가: Replaces *readfp()*.

read_string(*string*, *source='<string>'*)
Parse configuration data from a string.

Optional argument *source* specifies a context-specific name of the string passed. If not given, `'<string>'` is used. This should commonly be a filesystem path or a URL.

버전 3.2에 추가.

read_dict(*dictionary*, *source='<dict>'*)
Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument *source* specifies a context-specific name of the dictionary passed. If not given, `<dict>` is used.

This method can be used to copy state between parsers.

버전 3.2에 추가.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *DEFAULTSECT* in that order. If the key is not found and *fallback* is provided, it is used as a fallback value. `None` can be provided as a *fallback* value.

All the `'%'` interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

버전 3.2에서 변경: Arguments *raw*, *vars* and *fallback* are keyword only to protect users from trying to use the third argument as the *fallback* fallback (especially when using the mapping protocol).

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
A convenience method which coerces the *option* in the specified *section* to an integer. See *get()* for explanation of *raw*, *vars* and *fallback*.

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
A convenience method which coerces the *option* in the specified *section* to a floating point number. See *get()* for explanation of *raw*, *vars* and *fallback*.

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are `'1'`, `'yes'`, `'true'`, and `'on'`, which cause this method to return `True`, and `'0'`, `'no'`, `'false'`, and `'off'`, which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise *ValueError*. See *get()* for explanation of *raw*, *vars* and *fallback*.

items(*raw=False*, *vars=None*)
items(*section*, *raw=False*, *vars=None*)
When *section* is not given, return a list of *section_name*, *section_proxy* pairs, including DEFAULTSECT.

Otherwise, return a list of *name*, *value* pairs for the options in the given *section*. Optional arguments have the same meaning as for the *get()* method.

set(*section*, *option*, *value*)
If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. *option* and *value* must be strings; if not, *TypeError* is raised.

write(*fileobject*, *space_around_delimiters=True*)
Write a representation of the configuration to the specified *file object*, which must be opened in text mode (accepting strings). This representation can be parsed by a future *read()* call. If *space_around_delimiters* is true, delimiters between keys and values are surrounded by spaces.

remove_option(*section*, *option*)
Remove the specified *option* from the specified *section*. If the section does not exist, raise *NoSectionError*. If the option existed to be removed, return *True*; otherwise return *False*.

remove_section(*section*)
Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

optionxform(*option*)
Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to `str`, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before *optionxform()* is called.

readfp(*fp*, *filename=None*)
버전 3.2부터 폐지: Use *read_file()* instead.

버전 3.2에서 변경: *readfp()* now iterates on *fp* instead of calling `fp.readline()`.

For existing code calling *readfp()* with arguments which don't support iteration, the following generator may be used as a wrapper around the file-like object:

```
def readline_generator(fp):
    line = fp.readline()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    while line:
        yield line
        line = fp.readline()
```

Instead of `parser.readfp(fp)` use `parser.read_file(readline_generator(fp))`.

`configparser.`**`MAX_INTERPOLATION_DEPTH`**
The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only when the default *interpolation* is used.

14.2.10 RawConfigParser Objects

class `configparser.`**`RawConfigParser`**(*defaults=None*, *dict_type=collections.OrderedDict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*[, *interpolation*])
Legacy variant of the *ConfigParser*. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe `add_section` and `set` methods, as well as the legacy `defaults=` keyword argument handling.

참고: Consider using *ConfigParser* instead which checks types of the values to be stored internally. If you don't want interpolation, you can use `ConfigParser(interpolation=None)`.

---

add_section(*section*)
Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

set(*section*, *option*, *value*)
If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. While it is possible to use *RawConfigParser* (or *ConfigParser* with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

14.2.11 Exceptions

exception `configparser.`**`Error`**
Base class for all other *configparser* exceptions.

exception `configparser.`**`NoSectionError`**
Exception raised when a specified section is not found.

exception `configparser.`**`DuplicateSectionError`**
Exception raised if `add_section()` is called with the name of a section that is already present or in strict parsers when a section if found more than once in a single input file, string or dictionary.

버전 3.2에 추가: Optional `source` and `lineno` attributes and arguments to `__init__()` were added.

exception configparser.**DuplicateOptionError**
Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception configparser.**NoOptionError**
Exception raised when a specified option is not found in the specified section.

exception configparser.**InterpolationError**
Base class for exceptions raised when problems occur performing string interpolation.

exception configparser.**InterpolationDepthError**
Exception raised when string interpolation cannot be completed because the number of iterations exceeds *MAX_INTERPOLATION_DEPTH*. Subclass of *InterpolationError*.

exception configparser.**InterpolationMissingOptionError**
Exception raised when an option referenced from a value does not exist. Subclass of *InterpolationError*.

exception configparser.**InterpolationSyntaxError**
Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of *InterpolationError*.

exception configparser.**MissingSectionHeaderError**
Exception raised when attempting to parse a file which has no section headers.

exception configparser.**ParsingError**
Exception raised when errors occur attempting to parse a file.

버전 3.2에서 변경: The `filename` attribute and `__init__()` argument were renamed to `source` for consistency.

14.3 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*])
netrc 인스턴스나 서브 클래스 인스턴스는 netrc 파일의 데이터를 캡슐화합니다. 초기화 인자가 있으면 구문 분석할 파일을 지정합니다. 인자를 지정하지 않으면, *os.path.expanduser()*에 의해 결정된 사용자 홈 디렉터리에 있는 파일 `.netrc`를 읽습니다. 그렇지 않으면, *FileNotFoundError* 예외가 발생합니다. 구문 분석 에러는 파일 이름, 줄 번호 및 종료 토큰을 포함하는 진단 정보로 *NetrcParseError*를 발생시킵니다. POSIX 시스템에서 인자가 지정되지 않을 때, 파일 소유권이나 권한이 안전하지 않으면 (프로세스를 실행하는 사용자가 아닌 다른 사용자가 소유하거나 다른 모든 사용자가 읽기 또는 쓰기로 액세스할 수 있는 경우), `.netrc` 파일에 암호가 존재하면 *NetrcParseError*가 발생합니다. 이것은 ftp와 `.netrc`를 사용하는 다른 프로그램과 동등한 보안 행동을 구현합니다.

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.

exception netrc.**NetrcParseError**
소스 텍스트에 문법적인 에러가 있을 때 *netrc* 클래스에서 발생하는 예외. 이 예외 인스턴스는 세 가지

흥미로운 어트리뷰트를 제공합니다. `msg`는 에러의 텍스트 설명이고, `filename`은 소스 파일의 이름이며, `lineno`는 에러가 발견된 줄 번호입니다.

14.3.1 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 `None`을 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.

참고: 암호는 ASCII 문자 집합의 부분집합으로 제한됩니다. 모든 ASCII 구두점을 암호에 사용할 수 있지만, 공백과 인쇄 할 수 없는 문자는 암호에 사용할 수 없습니다. 이것은 .netrc 파일이 구문 분석되는 방식으로 인한 제한 사항이며 향후 제거될 수 있습니다.

14.4 xdrlib — XDR 데이터 인코딩과 디코딩

소스 코드: Lib/xdrlib.py

---

xdrlib 모듈은 1987년 6월에 Sun Microsystems, Inc.가 작성한 **RFC 1014**에 설명된 외부 데이터 표현 표준(External Data Representation Standard)을 지원합니다. 이 모듈은 RFC에 설명된 대부분의 데이터형을 지원합니다.

xdrlib 모듈은 두 개의 클래스를 정의합니다. 하나는 변수를 XDR 표현으로 패킹하고, 다른 하나는 XDR 표현으로부터 언패킹합니다. 또한, 두 가지 예외 클래스가 있습니다.

class xdrlib.**Packer**
*Packer*는 데이터를 XDR 표현으로 패킹하는 클래스입니다. *Packer* 클래스는 인자 없이 인스턴스화됩니다.

class xdrlib.**Unpacker**(*data*)
`Unpacker`는 문자열 버퍼에서 XDR 데이터값을 언패킹하는 반대 클래스입니다. 입력 버퍼는 *data*로 주어집니다.

더 보기:

RFC 1014 - XDR: External Data Representation Standard 이 RFC는 이 모듈이 처음 작성되었을 당시에 XDR이었던 데이터의 인코딩을 정의합니다. **RFC 1832**로 개정되었습니다.

RFC 1832 - XDR: External Data Representation Standard XDR의 개정된 정의를 제공하는 최신 RFC

14.4.1 Packer 객체

Packer 인스턴스에는 다음과 같은 메서드가 있습니다:

Packer.**get_buffer**()
현재의 팩 버퍼를 문자열로 반환합니다.

Packer.**reset**()
팩 버퍼를 빈 문자열로 재설정합니다.

일반적으로, 적절한 pack_type() 메서드를 호출하여 가장 자주 쓰이는 XDR 데이터형을 팩할 수 있습니다. 각 메서드는 팩할 값인 단일 인자를 취합니다. 다음과 같은 간단한 데이터형의 패킹 메서드가 지원됩니다: pack_uint(), pack_int(), pack_enum(), pack_bool(), pack_uhyper() 및 pack_hyper().

Packer.**pack_float**(*value*)
단정밀도 부동 소수점 숫자 *value*를 팩합니다.

Packer.**pack_double**(*value*)
배정밀도 부동 소수점 숫자 *value*를 팩합니다.

다음 메서드는 문자열, 바이트열 및 불투명 데이터의 패킹을 지원합니다:

Packer.**pack_fstring**(*n*, *s*)
고정 길이 문자열 *s*를 팩합니다. *n*는 문자열의 길이이지만 데이터 버퍼에 팩 되지는 않습니다. 4바이트 정렬을 보장하는 데 필요하면 문자열에 null 바이트가 채워집니다.

Packer.**pack_fopaque**(*n*, *data*)
*pack_fstring()*과 유사하게, 고정 길이의 불투명한 데이터 스트림을 팩합니다.

Packer.**pack_string**(*s*)
가변 길이 문자열 *s*를 팩합니다. 문자열의 길이를 먼저 부호 없는 정수로 팩하고, 문자열 데이터는 *pack_fstring()*으로 팩합니다.

Packer.**pack_opaque**(*data*)
*pack_string()*과 유사하게, 가변 길이 불투명 데이터 문자열을 팩합니다.

Packer.**pack_bytes**(*bytes*)
*pack_string()*과 유사하게, 가변 길이 바이트 스트림을 팩합니다.

다음 메서드는 배열과 리스트의 패킹을 지원합니다:

Packer.**pack_list**(*list*, *pack_item*)
균질한 항목의 *list*를 팩합니다. 이 메서드는 크기가 결정되지 않은 리스트에 유용합니다; 즉, 전체 리스트를 검사해볼 때까지 크기를 알 수 없습니다. 리스트의 각 항목에 대해 부호 없는 정수 1이 먼저 팩 되고, 그다음에 리스트로부터의 데이터값이 옵니다. *pack_item*은 개별 항목을 팩하려고 호출되는 함수입니다. 리스트의 끝에서 부호 없는 정수 0이 팩 됩니다.

예를 들어, 정수 리스트를 팩하려면, 이런 코드를 사용할 수 있습니다:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

Packer.**pack_farray**(*n*, *array*, *pack_item*)
균질한 항목의 고정 길이 리스트(*array*)를 팩합니다. *n*은 리스트의 길이입니다; 버퍼에 팩 되지 않지만, len(array)가 *n*과 같지 않으면 *ValueError* 예외가 발생합니다. 위와 같이, *pack_item*은 각 요소를 팩하는 데 사용되는 함수입니다.

Packer.**pack_array**(*list*, *pack_item*)
균질한 항목의 가변 길이 *list*를 팩합니다. 먼저, 리스트의 길이가 부호 없는 정수로 팩 되고, 각 요소는 위의 *pack_farray()*와 같이 팩 됩니다.

14.4.2 Unpacker 객체

Unpacker 클래스는 다음과 같은 메서드를 제공합니다:

Unpacker.**reset**(*data*)
지정된 *data*로 문자열 버퍼를 재설정합니다.

Unpacker.**get_position**()
데이터 버퍼의 현재의 언팩 위치를 반환합니다.

Unpacker.**set_position**(*position*)
데이터 버퍼 언팩 위치를 *position*으로 설정합니다. *get_position()*과 *set_position()* 사용 시 주의해야 합니다.

Unpacker.**get_buffer**()
현재의 언팩 데이터 버퍼를 문자열로 반환합니다.

Unpacker.**done**()
언팩 완료를 나타냅니다. 모든 데이터가 언팩 되지 않았으면 *Error* 예외를 발생시킵니다.

또한, *Packer*로 팩할 수 있는 모든 데이터형은 *Unpacker*로 언팩할 수 있습니다. 언패킹 메서드는 unpack_type() 형식이며 인자를 받아들이지 않습니다. 이것들은 언팩 된 객체를 반환합니다.

Unpacker.**unpack_float**()
단정밀도 부동 소수점 숫자를 언팩합니다.

Unpacker.**unpack_double**()
*unpack_float()*와 유사하게, 배정밀도 부동 소수점 숫자를 언팩합니다.

또한, 다음 메서드는 문자열, 바이트열 및 불투명 데이터를 언팩합니다:

Unpacker.**unpack_fstring**(*n*)
고정 길이 문자열을 언팩하고 반환합니다. *n*는 예상 문자 수입니다. 4바이트의 정렬을 보장하기 위해서, null 바이트로 채워졌다고 가정합니다.

Unpacker.**unpack_fopaque**(*n*)
*unpack_fstring()*과 유사하게, 고정 길이 불투명 데이터 스트림을 언팩하고 반환합니다.

Unpacker.**unpack_string**()
가변 길이 문자열을 언팩하고 반환합니다. 문자열의 길이를 먼저 부호 없는 정수로 언팩한 다음, 문자열 데이터를 *unpack_fstring()*으로 언팩합니다.

Unpacker.**unpack_opaque**()
*unpack_string()*과 유사하게, 가변 길이 불투명 데이터 문자열을 언팩하고 반환합니다.

Unpacker.**unpack_bytes**()
*unpack_string()*과 유사하게, 가변 길이 바이트 스트림을 언팩하고 반환합니다.

다음 메서드는 배열과 리스트의 언패킹을 지원합니다:

Unpacker.**unpack_list**(*unpack_item*)
균질한 항목의 리스트를 언팩하고 반환합니다. 리스트는 한 번에 한 요소씩 먼저 부호 없는 정수 플래그를 언팩해서 언팩합니다. 플래그가 1이면, 항목이 언팩되어 리스트에 추가됩니다. 0 플래그는 리스트의 끝을 나타냅니다. *unpack_item*은 항목을 언팩하는 함수입니다.

Unpacker.**unpack_farray**(*n*, *unpack_item*)
균질한 항목의 고정 길이 배열을 언팩하고 (리스트로) 반환합니다. *n*은 버퍼에서 예상되는 리스트 요소의 수입니다. 위와 같이, *unpack_item*은 각 요소를 언팩하는 데 사용되는 함수입니다.

Unpacker.**unpack_array**(*unpack_item*)
균질한 항목의 가변 길이 *list*를 언팩하고 반환합니다. 먼저, 리스트의 길이를 부호 없는 정수로 언팩하고, 각 요소는 위의 *unpack_farray()*처럼 언팩됩니다.

14.4.3 예외

이 모듈의 예외는 클래스 인스턴스로 코딩됩니다:

exception xdrlib.**Error**
베이스 예외 클래스. *Error*에는 에러에 대한 설명을 포함하는 공용 어트리뷰트 msg가 하나 있습니다.

exception xdrlib.**ConversionError**
*Error*에서 파생된 클래스. 추가 인스턴스 변수가 없습니다.

다음은 이러한 예외 중 하나를 잡는 방법의 예입니다:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```

14.5 plistlib — 맥 OS X .plist 파일 생성과 구문 분석

소스 코드: Lib/plistlib.py

이 모듈은 주로 Mac OS X에서 사용되는 "프로퍼티 리스트(property list)" 파일을 읽고 쓰는 인터페이스를 제공하며 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 바이트열 객체로 작업하려면, *dumps()*와 *loads()*를 사용하십시오.

값은 문자열, 정수, 부동 소수점, 논릿값, 튜플, 리스트, 딕셔너리 (단, 문자열 키만 가능), *Data*, *bytes*, bytesarray 또는 *datetime.datetime* 객체일 수 있습니다.

버전 3.4에서 변경: 새 API, 이전 API는 폐지되었습니다. 바이너리 형식 plist에 대한 지원이 추가되었습니다.

더 보기:

PList manual page 애플의 파일 형식 설명서.

이 모듈은 다음 함수를 정의합니다:

plistlib.**load**(*fp*, ***, *fmt=None*, *use_builtin_types=True*, *dict_type=dict*)
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

*use_builtin_types*가 참(기본값)이면 바이너리 데이터가 *bytes*의 인스턴스로 반환되고, 그렇지 않으면 *Data*의 인스턴스로 반환됩니다.

*dict_type*은 plist 파일에서 읽은 딕셔너리에 사용되는 형입니다.

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.

바이너리 형식의 구문 분석기는 파일을 구문 분석할 수 없을 때 `InvalidFileException`를 발생시킵니다.

버전 3.4에 추가.

plistlib.**loads**(*data*, **, fmt=None, use_builtin_types=True, dict_type=dict*)
바이트열 객체에서 plist를 로드합니다. 키워드 인자에 대한 설명은 *load()*를 참조하십시오.

버전 3.4에 추가.

plistlib.**dump**(*value*, *fp*, **, fmt=FMT_XML, sort_keys=True, skipkeys=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.

객체가 지원되지 않는 형이거나 지원되지 않는 형의 객체를 포함하는 컨테이너면 *TypeError*가 발생합니다.

(바이너리) plist 파일에서 표현할 수 없는 정숫값은 *OverflowError*를 발생시킵니다.

버전 3.4에 추가.

plistlib.**dumps**(*value*, **, fmt=FMT_XML, sort_keys=True, skipkeys=False*)
plist 형식의 바이트열 객체로 *value*를 반환합니다. 이 함수의 키워드 인자에 대한 설명은 *dump()* 설명서를 참조하십시오.

버전 3.4에 추가.

다음 함수는 폐지되었습니다:

plistlib.**readPlist**(*pathOrFile*)
plist 파일을 읽습니다. *pathOrFile*은 파일 이름이나 (읽기 가능한 바이너리) 파일 객체일 수 있습니다. 해독된 루트 객체를 반환합니다 (일반적으로 딕셔너리입니다).

이 함수는 *load()*를 호출하여 실제 작업을 수행합니다. 키워드 인자에 대한 설명은 그 함수 문서를 참조하십시오.

버전 3.4부터 폐지: 대신 *load()*를 사용하십시오.

버전 3.7에서 변경: 결과의 딕셔너리 값은 이제 평범한 dict입니다. 더는 이 딕셔너리의 항목을 액세스하기 위해 어트리뷰트 액세스를 사용할 수 없습니다.

plistlib.**writePlist**(*rootObject*, *pathOrFile*)
XML plist 파일에 *rootObject*를 씁니다. *pathOrFile*는 파일 이름이나 (쓰기 가능한 바이너리) 파일 객체일 수 있습니다

버전 3.4부터 폐지: 대신 *dump()*를 사용하십시오.

plistlib.**readPlistFromBytes**(*data*)
바이트열 객체에서 plist 데이터를 읽습니다. 루트 객체를 반환합니다.

키워드 인자에 대한 설명은 *load()*를 참조하십시오.

버전 3.4부터 폐지: 대신 *loads()*를 사용하십시오.

버전 3.7에서 변경: 결과의 딕셔너리 값은 이제 평범한 dict입니다. 더는 이 딕셔너리의 항목을 액세스하기 위해 어트리뷰트 액세스를 사용할 수 없습니다.

plistlib.**writePlistToBytes**(*rootObject*)
*rootObject*를 XML plist 형식의 바이트열 객체로 반환합니다.

버전 3.4부터 폐지: 대신 *dumps()*를 사용하십시오.

다음 클래스를 사용할 수 있습니다:

class plistlib.**Data**(*data*)
바이트열 객체 *data*를 감싸는 "데이터" 래퍼 객체를 반환합니다. plist 와 상호 변환하는 함수에서 plist 에서 사용할 수 있는 <data> 형을 나타내기 위해 사용됩니다.

하나의 어트리뷰트 data가 있는데, 저장된 파이썬 바이트열 객체를 조회하는 데 사용될 수 있습니다.

버전 3.4부터 폐지: 대신 *bytes* 객체를 사용하십시오.

다음 상수를 사용할 수 있습니다:

plistlib.**FMT_XML**
plist 파일의 XML 형식.

버전 3.4에 추가.

plistlib.**FMT_BINARY**
plist 파일의 바이너리 형식

버전 3.4에 추가.

14.5.1 예제

plist 만들기:

```
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.fromtimestamp(time.mktime(time.gmtime())),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

plist 구문 분석하기:

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
print(pl["aKey"])
```

CHAPTER 15

암호화 서비스

이 장에서 설명하는 모듈은 다양한 암호화 알고리즘을 구현합니다. 가용성은 설치에 달려있습니다. 유닉스 시스템에서는 *crypt* 모듈을 사용할 수도 있습니다. 다음은 개요입니다:

15.1 `hashlib` — Secure hashes and message digests

Source code: Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet **RFC 1321**). The terms "secure hash" and "message digest" are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

참고: If you want the adler32 or crc32 hash functions, they are available in the *zlib* module.

경고: Some algorithms have known hash collision weaknesses, refer to the "See also" section at the end.

15.1.1 Hash algorithms

There is one constructor method named for each type of *hash*. All return a hash object with the same simple interface. For example: use `sha256()` to create a SHA-256 hash object. You can now feed this object with *bytes-like objects* (normally *bytes*) using the `update()` method. At any point you can ask it for the *digest* of the concatenation of the data fed to it so far using the `digest()` or `hexdigest()` methods.

참고: For better multithreading performance, the Python *GIL* is released for data larger than 2047 bytes at object creation or on update.

참고: Feeding string objects into `update()` is not supported, as hashes work on bytes, not on characters.

Constructors for hash algorithms that are always present in this module are `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, *`blake2b()`*, and *`blake2s()`*. `md5()` is normally available as well, though it may be missing if you are using a rare "FIPS compliant" build of Python. Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform. On most platforms the `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()` are also available.

버전 3.6에 추가: SHA3 (Keccak) and SHAKE constructors `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`.

버전 3.6에 추가: *`blake2b()`* and *`blake2s()`* were added.

For example, to obtain the digest of the byte string `b'Nobody inspects the spammish repetition'`:

```
>>> 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.digest_size
32
>>> m.block_size
64
```

More condensed:

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

hashlib.**new**(*name*[, *data*])

Is a generic constructor that takes the string *name* of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than *`new()`* and should be preferred.

Using *`new()`* with an algorithm provided by OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Hashlib provides the following constant attributes:

hashlib.**algorithms_guaranteed**

A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms. Note that 'md5' is in this list despite some upstream vendors offering an odd "FIPS compliant" Python build that excludes it.

버전 3.2에 추가.

hashlib.**algorithms_available**
A set containing the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to *new()*. *algorithms_guaranteed* will always be a subset. The same algorithm may appear multiple times in this set under different names (thanks to OpenSSL).

버전 3.2에 추가.

The following values are provided as constant attributes of the hash objects returned by the constructors:

hash.**digest_size**
The size of the resulting hash in bytes.

hash.**block_size**
The internal block size of the hash algorithm in bytes.

A hash object has the following attributes:

hash.**name**
The canonical name of this hash, always lowercase and always suitable as a parameter to *new()* to create another hash of this type.

버전 3.4에서 변경: The name attribute has been present in CPython since its inception, but until Python 3.4 was not formally specified, so may not exist on some platforms.

A hash object has the following methods:

hash.**update**(*data*)
Update the hash object with the *bytes-like object*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

버전 3.1에서 변경: The Python GIL is released to allow other threads to run while hash updates on data larger than 2047 bytes is taking place when using hash algorithms supplied by OpenSSL.

hash.**digest**()
Return the digest of the data passed to the *update()* method so far. This is a bytes object of size *digest_size* which may contain bytes in the whole range from 0 to 255.

hash.**hexdigest**()
Like *digest()* except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

hash.**copy**()
Return a copy ("clone") of the hash object. This can be used to efficiently compute the digests of data sharing a common initial substring.

15.1.2 SHAKE variable length digests

The `shake_128()` and `shake_256()` algorithms provide variable length digests with length_in_bits//2 up to 128 or 256 bits of security. As such, their digest methods require a length. Maximum length is not limited by the SHAKE algorithm.

shake.**digest**(*length*)
Return the digest of the data passed to the `update()` method so far. This is a bytes object of size *length* which may contain bytes in the whole range from 0 to 255.

shake.**hexdigest**(*length*)
Like *digest()* except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

15.1.3 Key derivation

Key derivation and key stretching algorithms are designed for secure password hashing. Naive algorithms such as `sha1(password)` are not resistant against brute-force attacks. A good password hashing function must be tunable, slow, and include a salt.

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

The function provides PKCS#5 password-based key derivation function 2. It uses HMAC as pseudorandom function.

The string *hash_name* is the desired name of the hash digest algorithm for HMAC, e.g. 'sha1' or 'sha256'. *password* and *salt* are interpreted as buffers of bytes. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

The number of *iterations* should be chosen based on the hash algorithm and computing power. As of 2013, at least 100,000 iterations of SHA-256 are suggested.

dklen is the length of the derived key. If *dklen* is `None` then the digest size of the hash algorithm *hash_name* is used, e.g. 64 for SHA-512.

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

버전 3.4에 추가.

참고: A fast implementation of *pbkdf2_hmac* is available with OpenSSL. The Python implementation uses an inline version of `hmac`. It is about three times slower and doesn't release the GIL.

`hashlib.`**`scrypt`**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

The function provides scrypt password-based key derivation function as defined in **RFC 7914**.

password and *salt* must be *bytes-like objects*. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

n is the CPU/Memory cost factor, *r* the block size, *p* parallelization factor and *maxmem* limits memory (OpenSSL 1.1.0 defaults to 32 MiB). *dklen* is the length of the derived key.

Availability: OpenSSL 1.1+.

버전 3.6에 추가.

15.1.4 BLAKE2

BLAKE2 is a cryptographic hash function defined in **RFC 7693** that comes in two flavors:

- **BLAKE2b**, optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes,
- **BLAKE2s**, optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

BLAKE2 supports **keyed mode** (a faster and simpler replacement for HMAC), **salted hashing**, **personalization**, and **tree hashing**.

Hash objects from this module follow the API of standard library's `hashlib` objects.

Creating hash objects

New hash objects are created by calling constructor functions:

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*)

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*)

These functions return the corresponding hash objects for calculating BLAKE2b or BLAKE2s. They optionally take these general parameters:

- *data*: initial chunk of data to hash, which must be *bytes-like object*. It can be passed only as positional argument.
- *digest_size*: size of output digest in bytes.
- *key*: key for keyed hashing (up to 64 bytes for BLAKE2b, up to 32 bytes for BLAKE2s).
- *salt*: salt for randomized hashing (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).
- *person*: personalization string (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).

The following table shows limits for general parameters (in bytes):

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

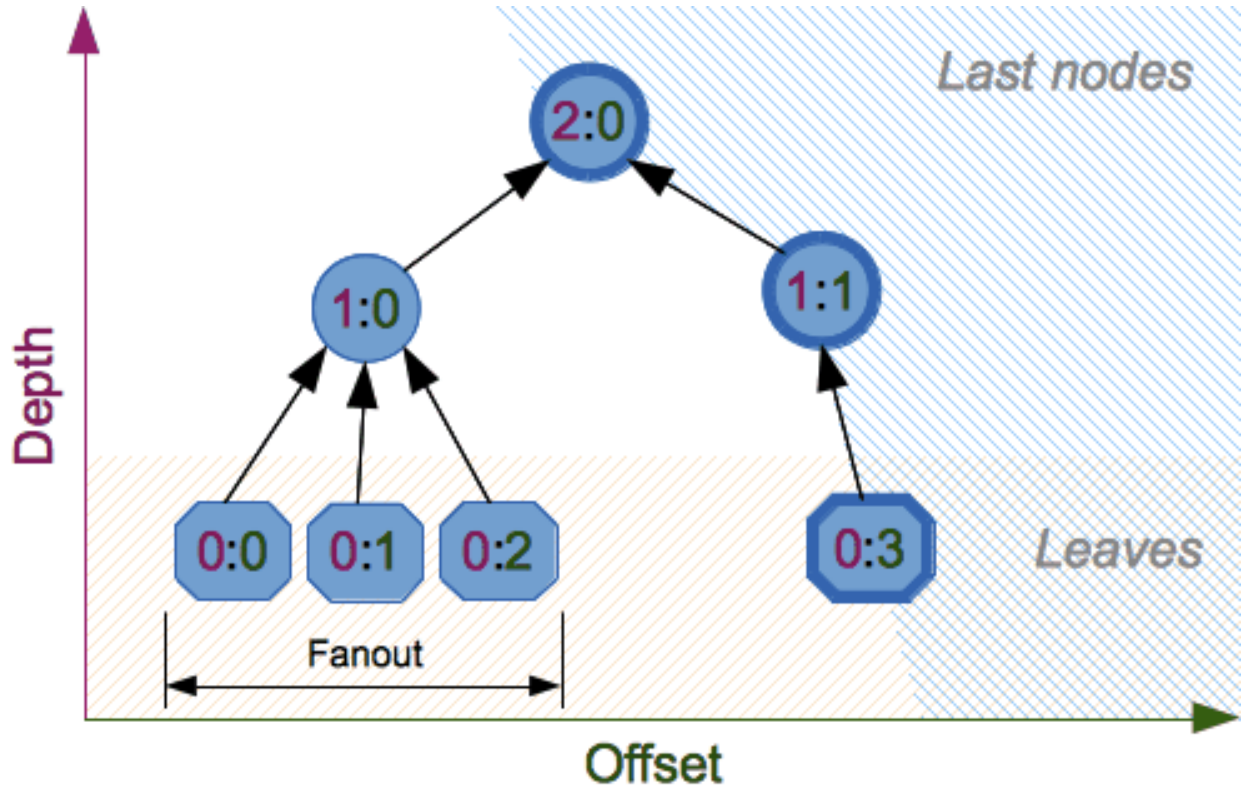
참고: BLAKE2 specification defines constant lengths for salt and personalization parameters, however, for convenience, this implementation accepts byte strings of any size up to the specified length. If the length of the parameter is less than specified, it is padded with zeros, thus, for example, `b'salt'` and `b'salt\x00'` is the same value. (This is not the case for *key*.)

These sizes are available as module *constants* described below.

Constructor functions also accept the following tree hashing parameters:

- *fanout*: fanout (0 to 255, 0 if unlimited, 1 in sequential mode).
- *depth*: maximal depth of tree (1 to 255, 255 if unlimited, 1 in sequential mode).
- *leaf_size*: maximal byte length of leaf (0 to 2**32-1, 0 if unlimited or in sequential mode).
- *node_offset*: node offset (0 to 2**64-1 for BLAKE2b, 0 to 2**48-1 for BLAKE2s, 0 for the first, leftmost, leaf, or in sequential mode).
- *node_depth*: node depth (0 to 255, 0 for leaves, or in sequential mode).
- *inner_size*: inner digest size (0 to 64 for BLAKE2b, 0 to 32 for BLAKE2s, 0 in sequential mode).
- *last_node*: boolean indicating whether the processed node is the last one (*False* for sequential mode).

See section 2.10 in BLAKE2 specification for comprehensive review of tree hashing.



Constants

blake2b.**SALT_SIZE**

blake2s.**SALT_SIZE**

Salt length (maximum length accepted by constructors).

blake2b.**PERSON_SIZE**

blake2s.**PERSON_SIZE**

Personalization string length (maximum length accepted by constructors).

blake2b.**MAX_KEY_SIZE**

blake2s.**MAX_KEY_SIZE**

Maximum key size.

blake2b.**MAX_DIGEST_SIZE**

blake2s.**MAX_DIGEST_SIZE**

Maximum digest size that the hash function can output.

Examples

Simple hashing

To calculate hash of some data, you should first construct a hash object by calling the appropriate constructor function (*blake2b()* or *blake2s()*), then update it with the data by calling `update()` on the object, and, finally, get the digest out of the object by calling `digest()` (or `hexdigest()` for hex-encoded string).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

As a shortcut, you can pass the first chunk of data to update directly to the constructor as the positional argument:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

You can call *hash.update()* as many times as you need to iteratively update the hash:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

Using different digest sizes

BLAKE2 has configurable size of digests up to 64 bytes for BLAKE2b and up to 32 bytes for BLAKE2s. For example, to replace SHA-1 with BLAKE2b without changing the size of output, we can tell BLAKE2b to produce 20-byte digests:

```
>>> 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
```

Hash objects with different digest sizes have completely different outputs (shorter hashes are *not* prefixes of longer hashes); BLAKE2b and BLAKE2s produce different outputs even if the output length is the same:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Keyed hashing

Keyed hashing can be used for authentication as a faster and simpler replacement for Hash-based message authentication code (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

This example shows how to get a (hex-encoded) 128-bit authentication code for message `b'message data'` with key `b'pseudorandom key'`:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

As a practical example, a web application can symmetrically sign cookies sent to users and later verify them to make sure they weren't tampered with:

```
>>> 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
```

Even though there's a native keyed hashing mode, BLAKE2 can, of course, be used in HMAC construction with *hmac* module:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Randomized hashing

By setting *salt* parameter users can introduce randomization to the hash function. Randomized hashing is useful for protecting against collision attacks on the hash function used in digital signatures.

> Randomized hashing is designed for situations where one party, the message preparer, generates all or part of a message to be signed by a second party, the message signer. If the message preparer is able to find cryptographic hash function collisions (i.e., two messages producing the same hash value), then they might prepare meaningful versions of the message that would produce the same hash value and digital signature, but with different results (e.g., transferring $1,000,000 to an account, rather than $10). Cryptographic hash functions have been designed with collision resistance as a major goal, but the current concentration on attacking cryptographic hash functions may result in a given cryptographic hash function providing less collision resistance than expected. Randomized hashing offers the signer additional protection by reducing the likelihood that a preparer can generate two or more messages that ultimately yield the same hash value during the digital signature generation process — even if it is practical to find collisions for the hash function. However, the use of randomized hashing may reduce the amount of security provided by a digital signature when all portions of the message are prepared by the signer.
>
> (NIST SP-800-106 "Randomized Hashing for Digital Signatures")

In BLAKE2 the salt is processed as a one-time input to the hash function during initialization, rather than as an input to each compression function.

경고: *Salted hashing* (or just hashing) with BLAKE2 or any other general-purpose cryptographic hash function, such as SHA-256, is not suitable for hashing passwords. See BLAKE2 FAQ for more information.

```
>>> 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
```

Personalization

Sometimes it is useful to force hash function to produce different digests for the same input for different purposes. Quoting the authors of the Skein hash function:

> We recommend that all application designers seriously consider doing this; we have seen many protocols where a hash that is computed in one part of the protocol can be used in an entirely different part because two hash computations were done on similar or related data, and the attacker can force the application to make the hash inputs the same. Personalizing each hash function used in the protocol summarily stops this type of attack.
>
> (The Skein Hash Function Family, p. 21)

BLAKE2 can be personalized by passing bytes to the *person* argument:

```
>>> 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'
```

Personalization together with the keyed mode can also be used to derive different keys from a single one.

```
>>> 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=
```

Tree mode

Here's an example of hashing a minimal tree with two leaf nodes:

```
  10
 /  \
00  01
```

This example uses 64-byte internal digests, and returns the 32-byte final digest:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... 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)
>>> # Right leaf
... 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)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

Credits

BLAKE2 was designed by *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn*, and *Christian Winnerlein* based on SHA-3 finalist BLAKE created by *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier*, and *Raphael C.-W. Phan*.

It uses core algorithm from ChaCha cipher designed by *Daniel J. Bernstein*.

The stdlib implementation is based on pyblake2 module. It was written by *Dmitry Chestnykh* based on C implementation written by *Samuel Neves*. The documentation was copied from pyblake2 and written by *Dmitry Chestnykh*.

The C code was partly rewritten for Python by *Christian Heimes*.

The following public domain dedication applies for both C hash function implementation, extension code, and this documentation:

> To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide. This software is distributed without any warranty.
>
> You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see https://creativecommons.org/publicdomain/zero/1.0/.

The following people have helped with development or contributed their changes to the project and the public domain according to the Creative Commons Public Domain Dedication 1.0 Universal:

- *Alexandr Sokolovskiy*

더 보기:

Module `hmac` A module to generate message authentication codes using hashes.

Module `base64` Another way to encode binary hashes for non-binary environments.

https://blake2.net Official BLAKE2 website.

https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf The FIPS 180-2 publication on Secure Hash Algorithms.

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms Wikipedia article with information on which algorithms have known issues and what that means regarding their use.

https://www.ietf.org/rfc/rfc2898.txt PKCS #5: Password-Based Cryptography Specification Version 2.0

15.2 hmac — 메시지 인증을 위한 키 해싱

소스 코드: Lib/hmac.py

이 모듈은 **RFC 2104**에서 설명한 대로 HMAC 알고리즘을 구현합니다.

hmac.**new**(*key*, *msg=None*, *digestmod=None*)
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 이름을 지원하며 기본값은 `hashlib.md5` 생성자입니다.

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.

Deprecated since version 3.4, will be removed in version 3.8: *digestmod*에 대한 묵시적 기본 다이제스트로서의 MD5는 폐지되었습니다.

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.

버전 3.7에 추가.

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).

버전 3.4에 추가.

HMAC.**name**
이 HMAC의 규범적 이름, 항상 소문자, 예를 들어 `hmac-md5`.

버전 3.4에 추가.

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.

버전 3.3에 추가.

더 보기:

모듈 *hashlib* 안전한 해시 함수를 제공하는 파이썬 모듈.

15.3 secrets — 비밀 관리를 위한 안전한 난수 생성

버전 3.6에 추가.

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**(*sequence*)
비어있지 않은 시퀀스로부터 무작위로 선택된 요소를 돌려줍니다.

secrets.**randbelow**(*n*)
범위 [0, *n*)에서 무작위 int를 돌려줍니다.

secrets.**randbits**(*k*)
k 무작위 비트를 가지는 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 None이거나 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 None이거나 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 None이거나 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 *a*와 *b*가 같으면 True를, 그렇지 않으면 False를 반환하는데, 타이밍 공격의 위험을 줄이는 방식을 사용합니다. 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
alphabet = string.ascii_letters + string.digits
password = ''.join(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
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(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:

```
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(choice(words) for i in range(4))
```

암호 복구 응용에 적합한 보안 토큰을 포함하는 추측하기 어려운 임시 URL을 생성합니다:

```
url = 'https://mydomain.com/reset=' + 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 — 기타 운영 체제 인터페이스

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.

참고: All functions in this module raise *OSError* (or subclasses thereof) in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

---

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*는 더 세분되어 있습니다. *os.uname()*은 시스템 종속 버전 정보를 제공합니다.

platform 모듈은 시스템의 아이덴티티에 대한 자세한 검사를 제공합니다.

16.1.1 파일명, 명령 줄 인자 및 환경 변수

파이썬에서는, 파일명, 명령 줄 인자 및 환경 변수가 문자열형을 사용하여 표시됩니다. 일부 시스템에서는, 운영 체제에 전달하기 전에 이러한 문자열을 바이트열로 인코딩하는 것이 필요합니다. 파이썬은 파일 시스템 인코딩을 사용하여 이 변환을 수행합니다 (*sys.getfilesystemencoding()*을 참조하세요).

버전 3.1에서 변경: 일부 시스템에서는, 파일 시스템 인코딩을 사용한 변환이 실패할 수 있습니다. 이때, 파이썬은 *surrogateescape* 인코딩 에러 처리기를 사용하는데, 디코딩할 때 디코딩 할 수 없는 바이트가 유니코드 문자 U+DCxx로 치환되고, 다시 인코딩할 때 원래 바이트로 변환됩니다.

파일 시스템 인코딩은 128보다 작은 모든 바이트를 성공적으로 디코딩함을 보장해야 합니다. 파일 시스템 인코딩이 이 보장을 제공하지 못하면, API 함수가 UnicodeError를 발생시킬 수 있습니다.

16.1.2 프로세스 매개 변수

이 함수들과 데이터 항목은 현재 프로세스와 사용자에 관한 정보와 관련 연산을 제공합니다.

os.**ctermid**()
프로세스의 제어 터미널에 해당하는 파일명을 반환합니다.

가용성: 유닉스.

os.**environ**
문자열 환경을 나타내는 매핑 객체입니다. 예를 들어, environ['HOME']은 홈 디렉터리의 경로명이며 (일부 플랫폼에서), C의 getenv("HOME")과 같습니다.

이 매핑은 *os* 모듈을 처음으로 임포트 할 때, 일반적으로 파이썬을 시작할 때 site.py를 처리하는 과정에서, 캡처됩니다. 이 시각 이후 변경된 환경은 os.environ을 직접 수정하여 변경한 경우를 제외하고는 os.environ에 반영되지 않습니다.

플랫폼이 *putenv()* 함수를 지원하면, 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.

참고: *putenv()*를 직접 호출해도 os.environ은 변경되지 않으므로, os.environ을 수정하는 것이 좋습니다.

참고: FreeBSD 및 맥 OS X를 포함한 일부 플랫폼에서, environ을 설정하면 메모리 누수가 발생할 수 있습니다. putenv()에 대한 시스템 설명서를 참조하십시오.

---

*putenv()*가 제공되지 않으면, 이 매핑의 수정된 복사본을 적절한 프로세스 생성 함수에 전달하여 자식 프로세스가 수정된 환경을 사용하게 할 수 있습니다.

플랫폼이 *unsetenv()* 기능을 지원하면, 이 매핑의 항목을 삭제하여 환경 변수를 삭제할 수 있습니다. 항목이 os.environ에서 삭제되거나, pop() 또는 clear() 메서드 중 하나가 호출되면 *unsetenv()*가 자동으로 호출됩니다.

os.**environb**
*environ*의 바이트열 버전: 환경을 바이트열로 나타내는 매핑 객체입니다. *environ*과 *environb*는 동기화됩니다 (*environb*를 수정하면 *environ*이 갱신되고, 그 반대도 마찬가지입니다).

environb is only available if *supports_bytes_environ* is True.

버전 3.2에 추가.

os.**chdir**(*path*)
os.**fchdir**(*fd*)
os.**getcwd**()
이 함수는 파일과 디렉터리에 설명되어 있습니다.

os.**fsencode**(*filename*)
'surrogateescape' 에러 처리기를, 또는 윈도우에서는 'strict'를, 사용하여 파일 시스템 인코딩으로 경로류 *filename*을 인코딩합니다; *bytes*를 변경하지 않고 반환합니다.

*fsdecode()*는 역 함수입니다.

버전 3.2에 추가.

버전 3.6에서 변경: *os.PathLike* 인터페이스를 구현하는 객체를 받아들이도록 지원이 추가되었습니다.

os.**fsdecode**(*filename*)
'surrogateescape' 에러 처리기를, 또는 윈도우에서는 'strict'를, 사용하여 파일 시스템 인코딩으로 경로류 *filename*을 디코딩합니다; *str*을 변경하지 않고 반환합니다.

*fsencode()*는 역 함수입니다.

버전 3.2에 추가.

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.

버전 3.6에 추가.

class os.**PathLike**
파일 시스템 경로를 나타내는 객체(예를 들어 *pathlib.PurePath*)의 추상 베이스 클래스입니다.

버전 3.6에 추가.

abstractmethod **__fspath__**()
객체의 파일 시스템 경로 표현을 돌려줍니다.

이 메서드는 *str*이나 *bytes* 객체만 반환해야 하며, *str*을 선호합니다.

os.**getenv**(*key*, *default=None*)
존재하면 환경 변수 *key*의 값을 반환하고, 그렇지 않으면 *default*를 반환합니다. *key*, *default* 및 결과는 str입니다.

유닉스에서, 키와 값은 *sys.getfilesystemencoding()*과 'surrogateescape' 에러 처리기로 디코딩됩니다. 다른 인코딩을 사용하려면 *os.getenvb()*를 사용하십시오.

가용성: 대부분의 유닉스, 윈도우.

os.**getenvb**(*key*, *default=None*)
존재하면 환경 변수 *key* 의 값을 반환하고, 그렇지 않으면 *default* 를 반환합니다. *key*, *default* 및 결과는 bytes입니다.

getenvb() is only available if *supports_bytes_environ* is `True`.

가용성: 대부분의 유닉스.

버전 3.2에 추가.

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.

버전 3.2에 추가.

os.**getegid**()
현재 프로세스의 유효(effective) 그룹 ID를 반환합니다. 이것은 현재 프로세스에서 실행 중인 파일의 "set id" 비트에 해당합니다.

가용성: 유닉스.

os.**geteuid**()
현재 프로세스의 유효(effective) 사용자 ID를 반환합니다.

가용성: 유닉스.

os.**getgid**()
현재 프로세스의 실제(real) 그룹 ID를 반환합니다.

가용성: 유닉스.

os.**getgrouplist**(*user*, *group*)
*user*가 속한 그룹 ID의 목록을 돌려줍니다. *group* 이 목록에 없으면 포함됩니다; 일반적으로 *group* 은 *user* 의 암호 레코드에서 그룹 ID 필드로 지정됩니다.

가용성: 유닉스.

버전 3.3에 추가.

os.**getgroups**()
현재 프로세스와 관련된 보충(supplemental) 그룹 ID 목록을 반환합니다.

가용성: 유닉스.

참고: 맥 OS X에서, *getgroups()* 동작은 다른 유닉스 플랫폼과 약간 다릅니다. 파이썬 인터프리터가 `10.5` 또는 이전 버전의 배포 대상으로 빌드되면, *getgroups()*는 현재 사용자 프로세스와 관련된 유효 그룹 ID 목록을 반환합니다; 이 목록은 시스템 정의된 항목 수(일반적으로 16)로 제한되며, 적절하게 권한이 부여된 경우 *setgroups()*를 호출하여 수정할 수 있습니다. `10.5`보다 큰 배포 대상으로 빌드되면, *getgroups()*는 프로세스의 유효 사용자 ID와 연관된 사용자에 대한 현재 그룹 액세스 목록을 반환합니다; 그룹 액세스 목록은 프로세스 수명 동안 변경될 수 있으며, *setgroups()* 호출의 영향을 받지 않고, 길이도 16개로 제한되지 않습니다. 배포 대상 값(`MACOSX_DEPLOYMENT_TARGET`)은 *sysconfig.get_config_var()*를 통해 얻을 수 있습니다.

---

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 검사하

여 사용자가 누구인지 알아내고, 현재 실제 사용자 ID의 로그인 이름을 얻기 위해 `pwd.getpwuid(os.getuid())[0]`로 폴백 하기 때문입니다.

가용성: 유닉스, 윈도우.

os.**getpgid**(*pid*)
프로세스 ID *pid* 를 갖는 프로세스의 프로세스 그룹 ID를 반환합니다. *pid* 가 0이면, 현재 프로세스의 프로세스 그룹 id가 반환됩니다.

가용성: 유닉스.

os.**getpgrp**()
현재 프로세스 그룹의 ID를 반환합니다.

가용성: 유닉스.

os.**getpid**()
현재의 프로세스 ID를 반환합니다.

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: 유닉스, 윈도우.

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: 유닉스.

버전 3.3에 추가.

os.**PRIO_PROCESS**
os.**PRIO_PGRP**
os.**PRIO_USER**
getpriority() 와 *setpriority()* 함수의 매개 변숫값

가용성: 유닉스.

버전 3.3에 추가.

os.**getresuid**()
현재 프로세스의 실제 (real), 유효 (effective) 및 저장된 (saved) 사용자 ID를 나타내는 튜플 (ruid, euid, suid)를 반환합니다.

가용성: 유닉스.

버전 3.2에 추가.

os.**getresgid**()
현재 프로세스의 실제 (real), 유효 (effective) 및 저장된 (saved) 그룹 ID를 나타내는 튜플 (rgid, egid, sgid)를 반환합니다.

가용성: 유닉스.

버전 3.2에 추가.

os.**getuid**()
현재 프로세스의 실제 (real) 사용자 ID를 반환합니다.

가용성: 유닉스.

os.**initgroups**(*username*, *gid*)
지정된 사용자 이름이 구성원인 모든 그룹과 지정된 그룹 ID로 구성된 그룹 액세스 목록을 초기화하기 위해 시스템 initgroups()를 호출합니다.

가용성: 유닉스.

버전 3.2에 추가.

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.

가용성: 대부분의 유닉스, 윈도우.

참고: FreeBSD 및 맥 OS X를 포함한 일부 플랫폼에서, environ를 설정하면 메모리 누수가 발생할 수 있습니다. putenv에 관한 시스템 설명서를 참조하십시오.

---

putenv() 가 지원되면, os.environ의 항목에 대한 대입이 *putenv()* 에 대한 해당 호출로 자동 변환됩니다. 그러나, *putenv()* 에 대한 호출은 os.environ을 갱신하지 않으므로, 실제로는 os.environ 항목에 대입하는 것이 좋습니다.

os.**setegid**(*egid*)
현재 프로세스의 유효 그룹 ID를 설정합니다.

가용성: 유닉스.

os.**seteuid**(*euid*)
현재 프로세스의 유효 사용자 ID를 설정합니다.

가용성: 유닉스.

os.**setgid**(*gid*)
현재 프로세스의 그룹 ID를 설정합니다.

가용성: 유닉스.

os.**setgroups**(*groups*)
현재 프로세스와 연관된 보충(supplemental) 그룹 ID의 목록을 *groups*로 설정합니다. *groups* 는 시퀀스 여야 하며, 각 요소는 그룹을 식별하는 정수여야 합니다. 이 연산은 대개 슈퍼 유저만 사용할 수 있습니다.

가용성: 유닉스.

참고: 맥 OS X에서 *groups* 의 길이는 시스템이 정의한 최대 유효 그룹 ID 수(일반적으로 16)를 초과할 수 없습니다. setgroups()를 호출해서 설정한 것과 같은 그룹 목록을 반환하지 않는 경우에 관해서는 *getgroups()* 설명서를 참조하십시오.

---

os.**setpgrp**()
구현된 (있기는 하다면) 버전에 따라 시스템 호출 setpgrp() 나 setpgrp(0, 0)을 호출합니다. 의미에 대해서는 유닉스 매뉴얼을 참조하십시오.

가용성: 유닉스.

os.**setpgid**(*pid*, *pgrp*)
프로세스 ID가 *pid* 인 프로세스의 프로세스 그룹 ID를 *pgrp* 로 설정하기 위해 시스템 호출 setpgid()를 호출합니다. 의미에 대해서는 유닉스 매뉴얼을 참조하십시오.

가용성: 유닉스.

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 -20에서 19 사이의 값입니다. 기본 우선순위는 0입니다; 우선순위가 낮으면 더 유리하게 스케줄 됩니다.

가용성: 유닉스.

버전 3.3에 추가.

os.**setregid**(*rgid*, *egid*)
현재 프로세스의 실제(real) 및 유효한(effective) 그룹 ID를 설정합니다.

가용성: 유닉스.

os.**setresgid**(*rgid*, *egid*, *sgid*)
현재 프로세스의 실제(real), 유효(effective) 및 저장된(saved) 그룹 ID를 설정합니다.

가용성: 유닉스.

버전 3.2에 추가.

os.**setresuid**(*ruid*, *euid*, *suid*)
현재 프로세스의 실제(real), 유효(effective) 및 저장된(saved) 사용자 ID를 설정합니다.

가용성: 유닉스.

버전 3.2에 추가.

os.**setreuid**(*ruid*, *euid*)
현재 프로세스의 실제(real) 및 유효(effective) 사용자 ID를 설정합니다.

가용성: 유닉스.

os.**getsid**(*pid*)
시스템 호출 `getsid()`를 호출합니다. 의미에 대해서는 유닉스 매뉴얼을 참조하십시오.

가용성: 유닉스.

os.**setsid**()
시스템 호출 `setsid()`를 호출합니다. 의미에 대해서는 유닉스 매뉴얼을 참조하십시오.

가용성: 유닉스.

os.**setuid**(*uid*)
현재 프로세스의 사용자 ID를 설정합니다.

가용성: 유닉스.

os.**strerror**(*code*)
에러 코드 *code*에 해당하는 에러 메시지를 반환합니다. 알 수 없는 에러 코드가 주어질 때 `strerror()`가 `NULL`을 반환하는 플랫폼에서, *ValueError*가 발생합니다.

os.**supports_bytes_environ**
환경의 원시 OS 형이 바이트열이면 `True` (예를 들어, 윈도우에서는 `False`).

버전 3.2에 추가.

os.**umask**(*mask*)
현재 숫자 umask를 설정하고 이전 umask를 반환합니다.

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.

가용성: 최근 유닉스.

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.

*unsetenv()*가 지원되면, os.environ의 항목 삭제가 자동으로 *unsetenv()*에 대한 해당 호출로 변환됩니다. 그러나 *unsetenv()*에 대한 호출은 os.environ을 갱신하지 않으므로, 실제로는 os.environ 항목을 삭제하는 것이 좋습니다.

가용성: 대부분의 유닉스.

16.1.3 파일 객체 생성

이 함수는 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.

16.1.4 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)
    except OSError:
        pass
```

os.**device_encoding**(*fd*)
*fd*와 연관된 장치가 터미널에 연결되어 있을 때 인코딩을 설명하는 문자열을 반환합니다; 그렇지 않으면 *None*을 반환합니다.

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.

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.

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.

가용성: 유닉스.

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.

가용성: 유닉스.

os.**fdatasync**(*fd*)
파일 기술자 *fd*로 주어진 파일을 디스크에 쓰도록 강제합니다. 메타 데이터를 갱신하도록 강제하지 않습니다.

가용성: 유닉스.

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: 유닉스.

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: 유닉스.

os.**fsync**(*fd*)
파일 기술자 *fd* 의 파일을 디스크에 쓰도록 강제합니다. 유닉스에서는, 네이티브 `fsync()` 함수를 호출합니다; 윈도우에서는, MS `_commit()` 함수.

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: 유닉스, 윈도우.

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.

가용성: 유닉스, 윈도우.

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: 유닉스.

버전 3.5에 추가.

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.

가용성: 유닉스.

버전 3.3에 추가.

os.**F_LOCK**
os.**F_TLOCK**
os.**F_ULOCK**
os.**F_TEST**
*lockf()*가 취할 조치를 지정하는 플래그.

가용성: 유닉스.

버전 3.3에 추가.

os.**lseek**(*fd*, *pos*, *how*)
파일 기술자 *fd* 의 현재 위치를 *how* 에 따라 달리 해석되는 위치 *pos*로 설정합니다: *SEEK_SET* 이나 0 이면 파일의 시작 부분을 기준으로 위치를 설정합니다; *SEEK_CUR*이나 1 이면 현재 위치를 기준으로 설정합니다; *SEEK_END* 나 2 면 파일의 끝을 기준으로 설정합니다. 새 커서 위치를 파일의 시작에서 따진 바이트로 반환합니다.

os.**SEEK_SET**
os.**SEEK_CUR**
os.**SEEK_END**
lseek() 함수의 매개 변수. 값은 각각 0, 1, 2입니다.

버전 3.3에 추가: 일부 운영 체제는 os.SEEK_HOLE 이나 os.SEEK_DATA와 같은 추가 값을 지원할 수 있습니다.

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.

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 3.3에 추가: *dir_fd* 인자

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_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 가상 터미널 쌍을 엽니다. 파일 기술자의 쌍 (master, slave) 를 반환하는데, 각각 pty와 tty입니다. 새 파일 기술자는 상속 불가능합니다. (약간) 더 이식성 있는 접근 방식을 사용하려면, *pty* 모듈을 사용하십시오.

가용성: 일부 유닉스.

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: 유닉스, 윈도우.

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: 일부 유닉스.

버전 3.3에 추가.

os.**posix_fallocate**(*fd*, *offset*, *len*)
*fd*로 지정된 파일이 *offset* 에서 시작하여 *len* 바이트 동안 계속되도록 충분한 디스크 공간을 할당합니다.

가용성: 유닉스.

버전 3.3에 추가.

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: 유닉스.

버전 3.3에 추가.

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: 유닉스.

버전 3.3에 추가.

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: 유닉스.

버전 3.3에 추가.

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: 리눅스 2.6.30 이상, FreeBSD 6.0 이상, OpenBSD 2.7 이상. flags를 사용하려면 리눅스 4.6 이상이 필요합니다.

버전 3.7에 추가.

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.

일부 데이터가 성공적으로 읽히면, 읽은 바이트 수를 반환합니다. 읽은 바이트가 없으면, -1을 반환하고 errno를 *errno.EAGAIN*로 설정합니다.

가용성: 리눅스 4.14 이상.

버전 3.7에 추가.

os.**RWF_HIPRI**
우선순위가 높은 읽기/쓰기. 블록 기반 파일 시스템이 장치의 폴링을 사용할 수 있게 하여, 지연은 짧아지지만, 추가 자원을 사용할 수 있습니다.

현재, 리눅스에서, 이 기능은 *O_DIRECT* 플래그를 사용하여 열린 파일 기술자에만 사용할 수 있습니다.

가용성: 리눅스 4.6 이상.

버전 3.7에 추가.

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: 유닉스.

버전 3.3에 추가.

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*)
buffers 내용을 파일 기술자 *fd*의 오프셋 *offset* 에 쓰고, 파일 오프셋은 변경되지 않은 채로 남겨 둡니다. *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.

flags 인자는 다음 플래그 중 0개 이상의 비트별 OR를 포함합니다:

- *RWF_DSYNC*
- *RWF_SYNC*

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: 리눅스 2.6.30 이상, FreeBSD 6.0 이상, OpenBSD 2.7 이상. flags를 사용하려면 리눅스 4.7 이상이 필요합니다.

버전 3.7에 추가.

os.**RWF_DSYNC**
O_DSYNC open(2) 플래그의 쓰기마다 지정할 수 있는 버전을 제공합니다. 이 플래그 효과는 시스템 호출로 기록된 데이터 범위에만 적용됩니다.

가용성: 리눅스 4.7 이상.

버전 3.7에 추가.

os.**RWF_SYNC**
O_SYNC open(2) 플래그의 쓰기마다 지정할 수 있는 버전을 제공합니다. 이 플래그 효과는 시스템 호출로 기록된 데이터 범위에만 적용됩니다.

가용성: 리눅스 4.7 이상.

버전 3.7에 추가.

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.**sendfile**(*out*, *in*, *offset*, *count*)
os.**sendfile**(*out*, *in*, *offset*, *count*[, *headers*][, *trailers*], *flags=0*)
파일 기술자 *in*에서 파일 기술자 *out* 으로 *offset*에서 시작하여 *count* 바이트를 복사합니다. 전송된 바이트 수를 반환합니다. EOF에 도달하면 0을 반환합니다.

첫 번째 함수 서명은 *sendfile()*를 정의하는 모든 플랫폼에서 지원됩니다.

리눅스에서, *offset*이 `None`으로 주어지면, *in*의 현재 위치에서 바이트를 읽고 *in*의 위치가 갱신됩니다.

두 번째 경우는 맥 OS X와 FreeBSD 에 사용될 수 있는데, *headers* 와 *trailers* 는 *in* 의 데이터가 기록되는 전후에 기록되는 버퍼의 임의의 시퀀스입니다. 첫 번째 경우와 같은 결과를 반환합니다.

맥 OS X 및 FreeBSD 에서, *count* 의 값 0은 *in* 의 끝에 도달할 때까지 보내도록 지정합니다.

모든 플랫폼은 *out* 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: 유닉스.

참고: *sendfile()*의 고수준 래퍼는, *socket.socket.sendfile()*을 보십시오.

버전 3.3에 추가.

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: 유닉스.

버전 3.5에 추가.

os.**SF_NODISKIO**
os.**SF_MNOWAIT**
os.**SF_SYNC**
구현이 지원하는 경우, *sendfile()* 함수에 대한 매개 변수입니다.

가용성: 유닉스.

버전 3.3에 추가.

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: 유닉스.

버전 3.3에 추가.

os.**tcgetpgrp**(*fd*)
fd(*os.open()*에 의해 반환된 것과 같은 열린 파일 기술자)에 의해 주어진 터미널과 관련된 프로세스 그룹을 반환합니다.

가용성: 유닉스.

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: 유닉스.

os.**ttyname**(*fd*)
파일 기술자 *fd*와 관련된 터미널 장치를 나타내는 문자열을 돌려줍니다. *fd* 가 터미널 장치와 연관되어 있지 않으면, 예외가 발생합니다.

가용성: 유닉스.

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.stderr*에 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: 유닉스.

버전 3.3에 추가.

터미널의 크기 조회하기

버전 3.3에 추가.

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, *OSError*가 발생합니다.

*shutil.get_terminal_size()*가 일반적으로 사용해야 하는 고수준 함수이며, `os.get_terminal_size`는 저수준 구현입니다.

가용성: 유닉스, 윈도우.

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

버전 3.4에 추가.

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.

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: 윈도우.

os.**set_handle_inheritable**(*handle*, *inheritable*)
지정된 핸들의 "상속 가능(inheritable)" 플래그를 설정합니다.

가용성: 윈도우.

16.1.5 파일과 디렉터리

일부 유닉스 플랫폼에서, 이 함수 중 많은 것들이 다음 기능 중 하나 이상을 지원합니다:

- **파일 기술자 지정:** 일부 함수의 경우, *path* 인자는 경로명을 제공하는 문자열뿐만 아니라 파일 기술자일 수도 있습니다. 그러면 그 함수는 기술자가 참조하는 파일에서 작동합니다. (POSIX 시스템에서, 파이썬은 함수의 f... 버전을 호출합니다.)

  *os.supports_fd*를 사용하여, 플랫폼에서 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.

- **디렉터리 기술자에 상대적인 경로:** *dir_fd* 가 None이 아니면, 디렉터리를 가리키는 파일 기술자여야 하며, 대상 경로는 상대 경로여야 합니다; 그러면 경로는 그 디렉터리에 상대적입니다. 절대 경로이면, *dir_fd* 는 무시됩니다. (POSIX 시스템에서, 파이썬은 함수의 ...at 또는 f...at 버전을 호출합니다.)

  *os.supports_dir_fd*를 사용하여, 플랫폼에서 *dir_fd*가 지원되는지를 확인할 수 있습니다. 사용할 수 없을 때, 사용하면 *NotImplementedError*를 발생시킵니다.

- **심볼릭 링크를 따르지 않음:** *follow_symlinks* 가 `False`고, 대상 경로의 마지막 요소가 심볼릭 링크면, 함수는 링크가 가리키는 파일 대신 심볼릭 링크 자체에 대해 작동합니다. (POSIX 시스템에서, 파이썬은 함수의 `l...` 버전을 호출합니다.)

  *os.supports_follow_symlinks*를 사용하여, 플랫폼에서 *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.

This function can raise *OSError* and subclasses such as *FileNotFoundError*, *PermissionError*, and *NotADirectoryError*.

버전 3.3에 추가: 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.

가용성: 유닉스.

버전 3.3에 추가: *follow_symlinks* 인자.

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: 윈도우가 *chmod()*를 지원하더라도, (`stat.S_IWRITE` 와 `stat.S_IREAD` 상수나 해당 정숫값을 통해) 파일의 읽기 전용 플래그만 설정할 수 있습니다. 다른 모든 비트는 무시됩니다.

버전 3.3에 추가: *path*를 열린 파일 기술자로 지정하는 지원과 *dir_fd* 및 *follow_symlinks* 인자가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

`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.

가용성: 유닉스.

버전 3.3에 추가: *path* 에 열린 파일 기술자를 지정하는 것과 *dir_fd* 및 *follow_symlinks* 인자에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 지원합니다.

`os.`**`chroot`**(*path*)

현재 프로세스의 루트 디렉터리를 *path*로 변경합니다.

가용성: 유닉스.

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.

가용성: 유닉스.

`os.`**`getcwd`**()

현재 작업 디렉터리를 나타내는 문자열을 반환합니다.

os.**getcwdb**()
현재 작업 디렉터리를 나타내는 바이트열을 반환합니다.

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.

가용성: 유닉스.

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.

가용성: 유닉스.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

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.

가용성: 유닉스.

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.

이 함수는 디렉터리 기술자에 상대적인 경로를 제공하기 위해 *src_dir_fd* 와/나 *dst_dir_fd* 를 지정하는 것과, 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

가용성: 유닉스, 윈도우.

버전 3.2에서 변경: 윈도우 지원이 추가되었습니다.

버전 3.3에 추가: *src_dir_fd* , *dst_dir_fd* 및 *follow_symlinks* 인자를 추가했습니다.

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.

*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 str이 됩니다.

이 함수는 또한 파일 기술자 지정을 지원할 수 있습니다; 파일 기술자는 디렉터리를 참조해야 합니다.

참고: str 파일명을 bytes로 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 3.3에 추가: *path*에 열린 파일 기술자를 지정하는 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**lstat**(*path*, *, *dir_fd=None*)
주어진 경로에 대해 lstat() 시스템 호출과 동등한 작업을 수행합니다. *stat()*와 유사하지만, 심볼릭 링크를 따르지 않습니다. *stat_result* 객체를 반환합니다.

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: *src* 및 *dst*로 경로류 객체를 받아들입니다.

os.**mkdir**(*path*, *mode=0o777*, *, *dir_fd=None*)
숫자 모드 *mode*로 *path* 라는 디렉터리를 만듭니다.

디렉터리가 이미 존재하면, *FileExistsError*가 발생합니다.

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.

이 기능은 디렉터리 기술자에 상대적인 경로도 지원할 수 있습니다.

임시 디렉터리를 만들 수도 있습니다; *tempfile* 모듈의 *tempfile.mkdtemp()* 함수를 참조하십시오.

버전 3.3에 추가: *dir_fd* 인자

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

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.

mode 매개 변수는 말단 디렉터리를 만들기 위해 *mkdir()*로 전달됩니다; 이것이 어떻게 해석되는지는 *mkdir() 설명*을 보십시오. 새로 만들어지는 부모 디렉터리들의 파일 권한 비트를 설정하려면, *makedirs()*를 호출하기 전에 umask를 설정할 수 있습니다. 이미 존재하는 부모 디렉터리의 파일 권한 비트는 변경되지 않습니다.

If *exist_ok* is False (the default), an *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.

버전 3.2에 추가: *exist_ok* 매개 변수.

버전 3.4.1에서 변경: 파이썬 3.4.1 이전에는, *exist_ok* 가 True이고 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: *mode* 인자는 더는 새로 만들어지는 중간 수준 디렉터리의 파일 권한 비트에 영향을 주지 않습니다.

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: 유닉스.

버전 3.3에 추가: *dir_fd* 인자

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: 유닉스.

버전 3.3에 추가: *dir_fd* 인자

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**major**(*device*)
원시 장치 번호(보통 `stat`의 `st_dev` 이나 `st_rdev` 어트리뷰트)에서 장치 주 번호를 추출합니다.

os.**minor**(*device*)
원시 장치 번호(보통 `stat`의 `st_dev` 이나 `st_rdev` 어트리뷰트)에서 장치 부 번호를 추출합니다.

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: 유닉스.

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: 유닉스.

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.

가용성: 유닉스, 윈도우.

버전 3.2에서 변경: 윈도우 6.0 (Vista) 심볼릭 링크에 대한 지원이 추가되었습니다.

버전 3.3에 추가: *dir_fd* 인자

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**remove**(*path*, *, *dir_fd=None*)
Remove (delete) the file *path*. If *path* is a directory, an `IsADirectoryError` is raised. Use `rmdir()` to remove directories.

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.

버전 3.3에 추가: *dir_fd* 인자

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.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)
Rename the file or directory *src* to *dst*. If *dst* exists, the operation will fail with an `OSError` subclass in a number of cases:

On Windows, if *dst* exists a `FileExistsError` is always raised.

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* it 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.

버전 3.3에 추가: *src_dir_fd* 및 *dst_dir_fd* 인자

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.

버전 3.6에서 변경: *old* 와 *new* 에 경로류 객체를 받아들입니다.

os.**replace**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)
파일 또는 디렉터리 *src*의 이름을 *dst*로 바꿉니다. *dst* 가 디렉터리면, *OSError*가 발생합니다. *dst* 가 존재하고 파일이면, 사용자에게 권한이 있을 때 자동으로 대체됩니다. *src* 와 *dst* 가 다른 파일 시스템에 있으면, 작업이 실패할 수 있습니다. 성공하면, 이름 바꾸기는 원자적 연산이 됩니다 (이것은 POSIX 요구 사항입니다).

이 함수는 디렉터리 기술자에 상대적인 경로를 제공하도록 *src_dir_fd* 와/나 *dst_dir_fd* 를 지정하는 것을 지원할 수 있습니다.

버전 3.3에 추가.

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, an *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.

버전 3.3에 추가: *dir_fd* 매개 변수

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.

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.

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.

버전 3.6에 추가.

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: 유닉스 기반 시스템에서, *scandir()*은 시스템의 opendir() 과 readdir() 함수를 사용합니다. 윈도우에서는, Win32 FindFirstFileW 와 FindNextFileW 함수를 사용합니다.

버전 3.5에 추가.

버전 3.6에 추가: 컨텍스트 관리자 프로토콜과 *close()* 메서드 대한 지원이 추가되었습니다. *scandir()* 이터레이터가 모두 소진되거나 명시적으로 닫히지 않으면 *ResourceWarning*가 파괴자에서 방출됩니다.

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 `True`를 반환합니다; 항목이 다른 종류의 파일이거나 다른 종류의 파일을 가리키면, 또는 더는 존재하지 않으면 `False`를 반환합니다.

follow_symlinks 가 `False`면, 이 항목이 디렉터리일 때만 (심볼릭 링크를 따르지 않고) `True`를 반환합니다; 항목이 다른 종류의 파일이거나 더는 존재하지 않으면 `False`를 반환합니다.

결과는 *follow_symlinks* 가 `True` 및 `False`일 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_UNKNOWN`를 반환하는 특정 유닉스 파일 시스템은 예외입니다. 항목이 심볼릭 링크면, *follow_symlinks* 가 `False`가 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 `True`를 반환합니다; 항목이 디렉터리 또는 다른 비 파일 항목이거나, 그런 것을 가리키거나, 더는 존재하지 않으면 `False`를 반환합니다.

follow_symlinks 가 `False`면, 이 항목이 파일일 때만 (심볼릭 링크를 따르지 않고) `True`를 반환합니다; 항목이 디렉터리 나 다른 비 파일 항목이거나 더는 존재하지 않으면 `False`를 반환합니다.

결과는 `os.DirEntry` 객체에 캐시 됩니다. 캐싱, 시스템 호출, 예외 발생은 *is_dir()*과 같습니다.

is_symlink()
이 항목이 심볼릭 링크면 (망가졌다 하더라도) `True`를 반환합니다; 항목이 디렉터리 나 어떤 종류의 파일이거나 더는 존재하지 않으면 `False`를 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_UNKNOWN`를 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.

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 심볼릭 링크일 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.

`os.DirEntry`와 *pathlib.Path*의 여러 어트리뷰트와 메서드 사이에는 좋은 일치가 있음에 유의하십시오. 특히, `name` 어트리뷰트는 `is_dir()`, `is_file()`, `is_symlink()` 및 `stat()` 메서드와 같은 의미가 있습니다.

버전 3.5에 추가.

버전 3.6에서 변경: *PathLike* 인터페이스에 대한 지원이 추가되었습니다. 윈도우에서 *bytes* 경로에 대한 지원이 추가되었습니다.

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=False를 추가하거나 *lstat()*를 사용하십시오.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

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)
>>> statinfo.st_size
264
```

더 보기:

fstat() 및 *lstat()* 함수.

버전 3.3에 추가: *dir_fd* 및 *follow_symlinks* 인자와 경로 대신 파일 기술자를 지정하는 것을 추가했습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

class os.**stat_result**

어트리뷰트가 stat 구조체의 멤버와 대략 일치하는 객체. *os.stat()*, *os.fstat()* 및 *os.lstat()*의 결과로 사용됩니다.

어트리뷰트:

st_mode
파일 모드: 파일 유형 및 파일 모드 비트 (사용 권한).

st_ino
플랫폼에 따라 다르지만, 0이 아니면, 지정된 값의 st_dev은 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
플랫폼에 따라 다릅니다:

- 유닉스에서 가장 최근의 메타 데이터 변경 시간,
- 윈도우에서 생성 시간, 단위는 초.

st_atime_ns
나노초 정수 단위의 가장 최근의 액세스 시간.

st_mtime_ns
나노초 정수 단위의 가장 최근의 내용 수정 시간.

st_ctime_ns
플랫폼에 따라 다릅니다:

- 유닉스에서 가장 최근의 메타 데이터 변경 시간,
- 윈도우에서 생성 시간, 단위는 나노초 정수.

참고: *st_atime*, *st_mtime* 및 *st_ctime* 어트리뷰트의 정확한 의미와 해상도는 운영 체제와 파일 시스템에 따라 다릅니다. 예를 들어, FAT 또는 FAT32 파일 시스템을 사용하는 윈도우 시스템에서, *st_mtime*은 2초 해상도를, *st_atime*는 단지 1일 해상도를 갖습니다. 자세한 내용은 운영 체제 설명서를 참조하십시오.

마찬가지로, *st_atime_ns*, *st_mtime_ns* 및 *st_ctime_ns*가 항상 나노초 단위로 표시되지만, 많은 시스템은 나노초 정밀도를 제공하지 않습니다. 나노초 정밀도를 제공하는 시스템에서, *st_atime*, *st_mtime* 및 *st_ctime*를 저장하는 데 사용되는 부동 소수점 객체는, 이 값을 모두 보존할 수 없으므로, 약간 부정확합니다. 정확한 타임스탬프가 필요하면, 항상 *st_atime_ns*, *st_mtime_ns* 및 *st_ctime_ns*를 사용해야 합니다.

(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.

st_birthtime
파일 생성 시간.

Solaris 및 파생 상품에서, 다음 어트리뷰트도 사용할 수 있습니다:

st_fstype
파일을 포함하는 파일 시스템의 유형을 고유하게 식별하는 문자열.

맥 OS 시스템에서는, 다음 어트리뷰트도 사용할 수 있습니다:

st_rsize
파일의 실제 크기.

st_creator
파일의 생성자.

st_type
파일 유형.

윈도우 시스템에서는, 다음 어트리뷰트도 사용할 수도 있습니다:

st_file_attributes
윈도우 파일 어트리뷰트: `GetFileInformationByHandle()`에 의해 반환된 `BY_HANDLE_FILE_INFORMATION` 구조체의 `dwFileAttributes` 멤버. *stat* 모듈의 `FILE_ATTRIBUTE_*` 상수를 참조하십시오.

표준 모듈 *stat*는 stat 구조체에서 정보를 추출하는 데 유용한 함수와 상수를 정의합니다. (윈도우에서는, 일부 항목에 더미 값이 채워집니다.)

이전 버전과의 호환성을 위해, *stat_result* 인스턴스는 stat 구조체의 가장 중요한 (그리고 이식성 있는) 멤버를 제공하는 최소 10개의 정수로 구성된 튜플로 액세스할 수도 있는데, *st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime* 순서입니다. 일부 구현에서는 끝에 더 많은 항목을 추가 할 수 있습니다. 이전 버전의 파이썬과의 호환성을 위해, *stat_result*에 튜플로 액세스하면 항상 정수가 반환됩니다.

버전 3.3에 추가: *st_atime_ns*, *st_mtime_ns* 및 *st_ctime_ns* 멤버가 추가되었습니다.

버전 3.5에 추가: 윈도우에서 *st_file_attributes* 멤버를 추가했습니다.

버전 3.5에서 변경: 윈도우는 이제 사용 가능할 때 파일 인덱스를 *st_ino*로 반환합니다.

버전 3.7에 추가: Solaris/파생 제품에 *st_fstype* 멤버를 추가했습니다.

os.**statvfs**(*path*)
주어진 경로에 대해 `statvfs()` 시스템 호출을 수행합니다. 반환 값은 주어진 경로의 파일 시스템을 설명하는 객체인데, 어트리뷰트가 `statvfs` 구조체의 멤버인 `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`에 해당합니다.

`f_flag` 어트리뷰트의 비트 플래그에 대해 두 개의 모듈 수준 상수가 정의됩니다: `ST_RDONLY`가 설정되면, 파일 시스템은 읽기 전용으로 마운트되었고, `ST_NOSUID`가 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: 유닉스.

버전 3.2에서 변경: `ST_RDONLY` 및 `ST_NOSUID` 상수가 추가되었습니다.

버전 3.3에 추가: *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 3.7에 추가: f_fsid 추가.

os.**supports_dir_fd**

os 모듈의 어떤 함수가 *dir_fd* 매개 변수의 사용을 허용하는지를 나타내는 *Set* 객체입니다. 플랫폼마다 다른 기능을 제공하며, 한 플랫폼에서 작동할 수 있는 옵션은 다른 플랫폼에서 지원되지 않을 수 있습니다. 일관성을 위해, *dir_fd* 를 지원하는 함수는 항상 매개 변수를 지정할 수 있도록 하지만, 기능을 실제로 사용할 수 없으면 예외를 발생시킵니다.

특정 함수가 *dir_fd* 매개 변수의 사용을 허용하는지를 확인하려면, supports_dir_fd에 in 연산자를 사용하십시오. 예를 들어, 이 표현식은 *os.stat()*의 *dir_fd* 매개 변수가 로컬에서 사용 가능한지를 판별합니다:

```
os.stat in os.supports_dir_fd
```

현재 *dir_fd* 매개 변수는 유닉스 플랫폼에서만 작동합니다; 어느 것도 윈도우에서 작동하지 않습니다.

버전 3.3에 추가.

os.**supports_effective_ids**

os 모듈의 어떤 함수가 *os.access()*의 *effective_ids* 매개 변수의 사용을 허용하지를 나타내는 *Set* 객체입니다. 로컬 플랫폼이 지원하면, 컬렉션에 *os.access()*가 포함되고, 그렇지 않으면 비어있게 됩니다.

*os.access()*에 *effective_ids* 매개 변수를 사용할 수 있는지 확인하려면, supports_effective_ids에 in 연산자를 사용하십시오. 예를 들면 다음과 같습니다:

```
os.access in os.supports_effective_ids
```

현재 *effective_ids* 는 유닉스 플랫폼에서만 작동합니다; 윈도우에서는 작동하지 않습니다.

버전 3.3에 추가.

os.**supports_fd**

os 모듈의 어떤 함수가 자신의 *path* 매개 변수에 열린 파일 기술자를 지정하는 것을 허용하는지를 나타내는 *Set* 객체입니다. 플랫폼마다 다른 기능을 제공하며, 한 플랫폼에서 작동할 수 있는 옵션은 다른 플랫폼에서 지원되지 않을 수 있습니다. 일관성을 위해, *fd* 를 지원하는 함수는 항상 매개 변수를 지정할 수 있도록 하지만, 기능을 실제로 사용할 수 없으면 예외를 발생시킵니다.

특정 함수가 *path* 매개 변수에 열린 파일 기술자를 지정할 수 있도록 허용하는지를 확인하려면, supports_fd에 in 연산자를 사용하십시오. 예를 들어, 이 표현식은 로컬 플랫폼에서 *os.chdir()*가 호출될 때 열린 파일 기술자를 받아들이는지를 판별합니다:

```
os.chdir in os.supports_fd
```

버전 3.3에 추가.

os.**supports_follow_symlinks**

os 모듈의 어떤 함수가 *follow_symlinks* 매개 변수의 사용을 허용하는지를 나타내는 *Set* 객체입니다. 플랫폼마다 다른 기능을 제공하며, 한 플랫폼에서 작동할 수 있는 옵션은 다른 플랫폼에서 지원되지 않을 수 있습니다. 일관성을 위해, *follow_symlinks* 를 지원하는 함수는 항상 매개 변수를 지정할 수 있도록 하지만, 기능을 실제로 사용할 수 없으면 예외를 발생시킵니다.

특정 함수가 *follow_symlinks* 매개 변수의 사용을 허용하는지를 확인하려면, supports_follow_symlinks에 in 연산자를 사용하십시오. 예를 들어, 이 표현식은 *os.stat()*의 *follow_symlinks* 매개 변수가 로컬에서 사용 가능한지를 판별합니다:

```
os.stat in os.supports_follow_symlinks
```

버전 3.3에 추가.

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.

심볼릭 링크 지원은 윈도우 6.0(Vista)에서 소개되었습니다. *symlink()*는 6.0 이전의 윈도우 버전에서 *NotImplementedError*를 발생시킵니다.

이 함수는 디렉터리 기술자에 상대적인 경로를 지원할 수 있습니다.

참고: 윈도우에서, 심볼릭 링크를 성공적으로 만들려면 *SeCreateSymbolicLinkPrivilege* 가 필요합니다. 이 권한은 보통 일반 사용자에게는 부여되지 않지만, 관리자 수준으로 권한을 상승시킬 수 있는 계정에서는 사용할 수 있습니다. 권한을 얻거나 응용 프로그램을 관리자로 실행하는 것은 심볼릭 링크를 성공적으로 만들 방법입니다.

권한이 없는 사용자가 함수를 호출하면 *OSError*가 발생합니다.

가용성: 유닉스, 윈도우.

버전 3.2에서 변경: 윈도우 6.0 (Vista) 심볼릭 링크에 대한 지원이 추가되었습니다.

버전 3.3에 추가: *dir_fd* 인자를 추가했으며, 이제 비 윈도우 플랫폼에서 *target_is_directory* 를 허용합니다.

버전 3.6에서 변경: *src* 및 *dst*로 경로류 객체를 받아들입니다.

os.**sync**()
디스크에 모든 것을 쓰도록 강제합니다.

가용성: 유닉스.

버전 3.3에 추가.

os.**truncate**(*path*, *length*)
최대 *length* 바이트가 되도록 *path*에 해당하는 파일을 자릅니다.

이 함수는 파일 기술자 지정을 지원할 수 있습니다.

가용성: 유닉스, 윈도우.

버전 3.3에 추가.

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.

버전 3.3에 추가: *dir_fd* 매개 변수

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 `None`이 아니면, `(atime, mtime)` 형식의 2-튜플이어야 하며, 각 멤버는 초를 나타내는 int 또는 float입니다.
- *times* 가 `None`이고 *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.

디렉터리가 *path* 로 제공될 수 있는지는 운영 체제가 디렉터리를 파일로 구현하는지에 따라 다릅니다 (예를 들어, 윈도우는 그렇지 않습니다). 여기서 설정한 정확한 시간은 운영 체제가 액세스 및 수정 시간을 기록하는 해상도에 따라 뒤따르는 *stat()* 호출에서 반환되지 않을 수 있음에 주의해야 합니다; *stat()*를 참조하세요. 정확한 시간을 보존하는 가장 좋은 방법은 *utime* 의 *ns* 매개 변수에 *os.stat()* 결과 객체의 *st_atime_ns* 및 *st_mtime_ns* 필드를 사용하는 것입니다.

이 함수는 파일 기술자 지정, 디렉터리 기술자에 상대적인 경로 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

버전 3.3에 추가: *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 는 디렉터리 경로인 문자열입니다. *dirnames* 는 *dirpath* 의 하위 디렉터리 이름 리스트입니다(`'.'` 및 `'..'` 제외). *filenames* 는 *dirpath*에 있는 디렉터리가 아닌 파일의 이름 리스트입니다. 리스트에 들어있는 이름에는 경로 구성 요소가 들어 있지 않음에 유의하십시오. *dirpath* 에 있는 파일이나 디렉터리에 대한 전체 경로(*top*으로 시작하는)를 얻으려면, `os.path.join(dirpath, name)`을 수행하십시오.

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.

이 예는 시작 디렉터리 아래의 각 디렉터리에 있는 비 디렉터리 파일이 차지한 바이트 수를 표시합니다. 단, CVS 하위 디렉터리 아래는 보지 않습니다:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS 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
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))
```

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_fd`를 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 `False`임에 주의하십시오.

참고: *fwalk()*는 다음 이터레이션 단계까지만 유효한 파일 기술자를 산출하기 때문에, 더 오래 유지하려면 복제해야 합니다 (예를 들어, *dup()*로).

이 예는 시작 디렉터리 아래의 각 디렉터리에 있는 비 디렉터리 파일이 차지한 바이트 수를 표시합니다. 단, CVS 하위 디렉터리 아래는 보지 않습니다:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS 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)
```

가용성: 유닉스.

버전 3.3에 추가.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.7에서 변경: *bytes* 경로에 대한 지원이 추가되었습니다.

리눅스 확장 어트리뷰트

버전 3.3에 추가.

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.

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 None이면, *listxattr()*는 현재 디렉터리를 검사합니다.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)

path 에서 확장 파일 시스템 어트리뷰트 *attribute* 을 제거합니다. *attribute* 는 bytes 또는 str(직접 또는 *PathLike* 인터페이스를 통해 간접적으로) 이어야합니다. 문자열이면, 파일 시스템 인코딩으로 인코딩됩니다.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

버전 3.6에서 변경: *path* 및 *attribute*에 대해 경로류 객체를 받아들입니다.

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, ***, *follow_symlinks=True*)
path 에 있는 확장 파일 시스템 어트리뷰트 *attribute*를 *value*로 설정합니다. *attribute* 는 내장된 NUL이 없는 bytes 또는 str(직접 또는 *PathLike* 인터페이스를 통해 간접적으로)이어야 합니다. str이면, 파일 시스템 인코딩으로 인코딩됩니다. *flags* 는 *XATTR_REPLACE* 또는 *XATTR_CREATE* 일 수 있습니다. *XATTR_REPLACE*가 주어지고 어트리뷰트가 존재하지 않으면, EEXISTS가 발생합니다. *XATTR_CREATE*가 주어지고 어트리뷰트가 이미 존재하면, 어트리뷰트는 만들어지지 않고 ENODATA가 발생합니다.

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.

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.

16.1.6 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 `bar`만 인쇄합니다; `foo`는 무시되는 것처럼 보이게 됩니다.

os.**abort**()
현재 프로세스에 SIGABRT 시그널을 생성합니다. 유닉스에서, 기본 동작은 코어 덤프를 생성하는 것입니다; 윈도우에서, 프로세스는 즉시 종료 코드 `3`을 반환합니다. 이 함수를 호출하면 *signal.signal()*를 사용하여 SIGABRT에 등록된 파이썬 시그널 처리기를 호출하지 않게 됨에 주의하시기 바랍니다.

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.

*exec** 함수의 "l" 및 "v" 변형은 명령 줄 인자가 전달되는 방식이 다릅니다. "l" 변형은 아마도 코드가 작성될 때 매개 변수의 수가 고정되어 있다면 가장 작업하기 쉬운 것입니다; 개별 매개 변수는 단순히 `execl*()` 함수에 대한 추가 매개 변수가 됩니다. "v" 변형은 매개 변수의 개수가 가변적일 때 좋으며, 리스트나 튜플에 들어있는 인자가 *args* 매개 변수로 전달됩니다. 두 경우 모두, 자식 프로세스에 대한 인자는 실행 중인 명령의 이름으로 시작해야 하지만, 강제되지는 않습니다.

끝 근처에 "p"가 포함된 변형(*execlp()*, *execlpe()*, *execvp()* 및 *execvpe()*)은 `PATH` 환경 변수를 사용하여 프로그램 *file* 을 찾습니다. 환경이 대체 될 때 (다음 단락에서 설명할 *exec*e* 변형 중 하나를 사용하여), 새 환경이 `PATH` 변수의 소스로 사용됩니다. 다른 변형 *execl()*, *execle()*, *execv()* 및 *execve()*는 `PATH` 변수를 사용하여 실행 파일을 찾지 않습니다; *path* 에는 반드시 적절한 절대 또는 상대 경로가 있어야 합니다.

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.

가용성: 유닉스, 윈도우.

버전 3.3에 추가: *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: 종료하는 표준 방법은 `sys.exit(n)` 입니다. *_exit()*는 일반적으로 *fork()* 이후의 자식 프로세스에서만 사용해야 합니다.

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`
에러가 발생하지 않았음을 나타내는 종료 코드.

가용성: 유닉스.

`os.EX_USAGE`
잘못된 개수의 인자가 제공된 경우처럼, 명령이 잘못 사용되었음을 나타내는 종료 코드.

가용성: 유닉스.

`os.EX_DATAERR`
입력 데이터가 잘못되었음을 나타내는 종료 코드.

가용성: 유닉스.

`os.EX_NOINPUT`
입력 파일이 없거나 읽을 수 없음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_NOUSER**
지정된 사용자가 존재하지 않음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_NOHOST**
지정된 호스트가 존재하지 않음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_UNAVAILABLE**
필수 서비스를 사용할 수 없음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_SOFTWARE**
내부 소프트웨어 에러가 감지되었음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_OSERR**
포크 하거나 파이프를 만들 수 없는 등, 운영 체제 에러가 감지되었음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_OSFILE**
일부 시스템 파일이 없거나, 열 수 없거나, 다른 에러가 있음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_CANTCREAT**
사용자가 지정한 출력 파일을 만들 수 없음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_IOERR**
일부 파일에서 I/O를 수행하는 동안 에러가 발생했음을 나타내는 종료 코드.

가용성: 유닉스.

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: 유닉스.

os.**EX_PROTOCOL**
프로토콜 교환이 불법이거나 유효하지 않거나 이해되지 않았음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_NOPERM**
작업을 수행할 수 있는 권한이 충분하지 않음을 나타내는 종료 코드 (파일 시스템 문제에는 사용하지 않습니다).

가용성: 유닉스.

os.**EX_CONFIG**
어떤 종류의 구성 에러가 발생했음을 나타내는 종료 코드.

가용성: 유닉스.

os.**EX_NOTFOUND**
"항목을 찾을 수 없습니다" 와 같은 것을 의미하는 종료 코드.

가용성: 유닉스.

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.

> **경고:** fork()와 함께 SSL 모듈을 사용하는 응용 프로그램의 경우 *ssl*를 참조하십시오.

가용성: 유닉스.

os.**forkpty**()
새 의사 터미널을 자식의 제어 터미널로 사용하여 자식 프로세스를 포크 합니다. `(pid, fd)` 쌍을 반환하는데, 여기서 *pid* 는 자식에서 0이고, 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.

가용성: 일부 유닉스.

os.**kill**(*pid*, *sig*)
프로세스 *pid*에 시그널 *sig*를 보냅니다. 호스트 플랫폼에서 사용할 수 있는 구체적인 시그널에 대한 상수는 *signal* 모듈에 정의되어 있습니다.

윈도우: *signal.CTRL_C_EVENT* 및 *signal.CTRL_BREAK_EVENT* 시그널은 같은 콘솔 창을 공유하는 콘솔 프로세스(예를 들어, 일부 자식 프로세스)로만 보낼 수 있는 특수 시그널입니다. *sig*에 대한 다른 값은, 프로세스가 TerminateProcess API에 의해 무조건 종료되게 하고, 종료 코드는 *sig* 로 설정됩니다. 윈도우 버전의 *kill()*은 종료시킬 프로세스 핸들도 받아들입니다.

*signal.pthread_kill()*도 참조하십시오.

버전 3.2에 추가: 윈도우 지원.

os.**killpg**(*pgid*, *sig*)
시그널 *sig* 를 프로세스 그룹 *pgid* 로 보냅니다.

가용성: 유닉스.

os.**nice**(*increment*)
프로세스의 "우선도(niceness)"에 *increment* 을 추가합니다. 새로운 우선도를 반환합니다.

가용성: 유닉스.

os.**plock**(*op*)
프로그램 세그먼트를 메모리에 잠급니다. (<sys/lock.h>에서 정의된) *op* 값은 잠기는 세그먼트를 판별합니다.

가용성: 유닉스.

os.**popen**(*cmd*, *mode='r'*, *buffering=-1*)
명령 *cmd*와의 파이프 연결을 엽니다. 반환 값은 파이프에 연결된 열린 파일 객체이며, *mode* 가 `'r'` (기본값) 인지 ``'w'`` 인지에 따라 읽거나 쓸 수 있습니다. *buffering* 인자는 내장 *open()* 함수에서와 같은 의미가 있습니다. 반환된 파일 객체는 바이트열이 아닌 텍스트 문자열을 읽거나 씁니다.

`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.

이것은 *subprocess.Popen*를 사용하여 구현됩니다; 자식 프로세스를 관리하고 통신하는 보다 강력한 방법에 대해서는 이 클래스의 설명서를 참조하십시오.

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: 유닉스.

버전 3.7에 추가.

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 *Replacing Older Functions with the subprocess Module* 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.

*spawn** 함수의 "l" 및 "v" 변형은 명령 줄 인자가 전달되는 방식이 다릅니다. "l" 변형은 아마도 코드가 작성될 때 매개 변수의 수가 고정되어 있다면 가장 작업하기 쉬운 것입니다; 개별 매개 변수는 단순히 `spawnl*()` 함수에 대한 추가 매개 변수가 됩니다. "v" 변형은 매개 변수의 개수가 가변적일 때 좋으며, 리스트나 튜플에 들어있는 인자가 *args* 매개 변수로 전달됩니다. 두 경우 모두, 자식 프로세스에 대한 인자는 반드시 실행 중인 명령의 이름으로 시작해야 합니다.

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 127로 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)
```

가용성: 유닉스, 윈도우. *spawnlp()*, *spawnlpe()*, *spawnvp()*, *spawnvpe()*는 윈도우에서 사용할 수 없습니다. *spawnle()*와 *spawnve()*는 윈도우에서 스레드 안전하지 않습니다; 대신 *subprocess* 모듈을 사용하도록 권고합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**P_NOWAIT**
os.**P_NOWAITO**
*spawn** 계열 함수의 *mode* 매개 변수에 사용할 수 있는 값. 이 값 중 하나가 주어지면, spawn*() 함수는 새로운 프로세스가 생성되자마자 프로세스 ID를 반환 값으로 사용하여 반환됩니다.

가용성: 유닉스, 윈도우.

os.**P_WAIT**
*spawn** 계열 함수의 *mode* 매개 변수에 사용할 수 있는 값. 이것이 *mode* 로 주어지면, spawn*() 함수는 새 프로세스가 완료될 때까지 반환되지 않고, 실행이 성공한 프로세스의 종료 코드를 반환하거나, 시그널이 프로세스를 죽이면 -signal을 반환합니다.

가용성: 유닉스, 윈도우.

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: 윈도우.

os.**startfile**(*path*[, *operation*])
연관된 응용 프로그램으로 파일을 시작합니다.

operation 이 지정되지 않았거나 'open'이면, 윈도우 탐색기에서 파일을 두 번 클릭하거나, 대화형 명령 셸에서 **start** 명령에 인자로 파일명을 지정하는 것과 같은 역할을 합니다: 파일의 확장자와 연관된 (있다면) 응용 프로그램으로 파일이 열립니다.

다른 *operation* 이 주어지면, 파일로 수행해야 할 작업을 지정하는 "명령 동사"여야 합니다. 마이크로소프트에서 문서화 한 일반적인 동사는 'print' 와 'edit' (파일에 사용됨) 및 'explore' 와 'find' (디렉터리에 사용됨) 입니다.

*startfile()*는 연관된 응용 프로그램이 시작되자마자 반환합니다. 응용 프로그램이 닫히기를 기다리는 옵션과 응용 프로그램의 종료 상태를 검색할 방법이 없습니다. *path* 매개 변수는 현재 디렉터리에 상대적입니다. 절대 경로를 사용하려면 첫 번째 문자가 슬래시('/')가 아닌지 확인하십시오; 하부 Win32 ShellExecute() 함수는 첫 번째 문자가 슬래시면 작동하지 않습니다. *os.path.normpath()* 함수를 사용하여 경로가 Win32 용으로 올바르게 인코딩되도록 하십시오.

인터프리터 시작 오버헤드를 줄이기 위해, Win32 ShellExecute() 함수는 이 함수가 처음 호출될 때까지 결정(resolve)되지 않습니다. 함수를 결정할 수 없으면 *NotImplementedError*가 발생합니다.

가용성: 윈도우.

os.**system**(*command*)
서브 셸에서 명령(문자열)을 실행합니다. 이것은 표준 C 함수 system()를 호출하여 구현되며, 같은 제한이 있습니다. *sys.stdin* 등의 변경 사항은 실행된 명령의 환경에 반영되지 않습니다. *command*가 출력을 생성하면, 인터프리터 표준 출력 스트림으로 전송됩니다.

유닉스에서, 반환 값은 *wait()*에 지정된 형식으로 인코딩된 프로세스의 종료 상태입니다. POSIX는 C system() 함수의 반환 값의 의미를 지정하지 않으므로, 파이썬 함수의 반환 값은 시스템 종속적입니다.

윈도우에서, 반환 값은 *command*를 실행한 후 시스템 셸에서 반환한 값입니다. 셸은 윈도우 환경 변수 COMSPEC에 의해 제공됩니다: 보통 **cmd.exe**인데, 명령 실행의 종료 상태를 반환합니다; 기본이 아닌 셸을 사용하는 시스템에서는 셸 설명서를 참조하십시오.

subprocess 모듈은 새 프로세스를 생성하고 결과를 조회하는데, 더욱 강력한 기능을 제공합니다; 이 모듈을 사용하는 것이 이 함수들을 사용하는 것보다 더 바람직합니다. *subprocess* 설명서의 *Replacing Older Functions with the subprocess Module* 섹션에서 유용한 조리법을 확인하십시오.

가용성: 유닉스, 윈도우.

os.**times**()
현재 전역 프로세스 시간을 반환합니다. 반환 값은 5가지 어트리뷰트를 가진 객체입니다:

- user - 사용자 시간
- *system* - 시스템 시간
- children_user - 모든 자식 프로세스의 사용자 시간
- children_system - 모든 자식 프로세스의 시스템 시간
- elapsed - 과거의 고정된 시점 이후 실제 경과 시간

과거 호환성을 위해, 이 객체는 user, *system*, children_user, children_system 및 elapsed가 이 순서로 포함된 5-튜플처럼 작동합니다.

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: 유닉스, 윈도우.

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.

가용성: 유닉스.

os.**waitid**(*idtype*, *id*, *options*)
하나 이상의 자식 프로세스가 완료될 때까지 기다립니다. *idtype* 은 *P_PID*, *P_PGID* 또는 *P_ALL*이 될 수 있습니다. *id* 는 기다릴 pid를 지정합니다. *options*는 하나 이상의 *WEXITED*, *WSTOPPED* 또는 *WCONTINUED*의 OR로 구성되며, 추가로 *WNOHANG* 또는 *WNOWAIT*와 OR 될 수 있습니다. 반환 값은 siginfo_t 구조체에 포함된 데이터(즉, si_pid, si_uid, si_signo, si_status, si_code)를 나타내는 객체이거나, *WNOHANG*가 지정되고 대기 가능한 상태의 자식이 없으면 None입니다.

가용성: 유닉스.

버전 3.3에 추가.

os.**P_PID**
os.**P_PGID**
os.**P_ALL**
이것들은 *waitid()*의 *idtype* 에 사용 가능한 값입니다. *id* 가 어떻게 해석되는지에 영향을 미칩니다.

가용성: 유닉스.

버전 3.3에 추가.

os.**WEXITED**
os.**WSTOPPED**
os.**WNOWAIT**
기다릴 자식 시그널을 지정하는, *waitid()*의 *options* 에서 사용할 수 있는 플래그.

가용성: 유닉스.

버전 3.3에 추가.

os.**CLD_EXITED**
os.**CLD_DUMPED**
os.**CLD_TRAPPED**
os.**CLD_CONTINUED**
이것은 *waitid()*에 의해 반환된 결과에서 `si_code`의 가능한 값입니다.

가용성: 유닉스.

버전 3.3에 추가.

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.

시스템 호출이 -1을 반환하면, *OSError*가 errno 값으로 발생합니다.

윈도우에서: 프로세스 핸들 *pid*로 지정된 프로세스가 완료될 때까지 기다리고, *pid*와 종료 상태를 8비트 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.

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*)
*waitpid()*와 비슷하지만, 프로세스 ID 인자가 제공되지 않고 자식 프로세스 ID, 종료 상태 표시 및 자원 사용 정보가 포함된 3-요소 튜플이 반환된다는 점이 다릅니다. 자원 사용 정보에 대한 자세한 내용은 *resource.getrusage()*를 참조하십시오. 옵션 인자는 *waitpid()* 및 *wait4()*에 제공된 인자와 같습니다.

가용성: 유닉스.

os.**wait4**(*pid*, *options*)
*waitpid()*와 비슷하지만, 자식 프로세스 ID, 종료 상태 표시 및 자원 사용 정보가 포함된 3-요소 튜플이 반환된다는 점이 다릅니다. 자원 사용 정보에 대한 자세한 내용은 *resource.getrusage()*를 참조하십시오. *wait4()*의 인자는 *waitpid()*와 같습니다.

가용성: 유닉스.

os.**WNOHANG**
자식 프로세스 상태를 즉시 사용할 수 없으면, *waitpid()*가 즉시 반환하는 옵션입니다. 이 경우 이 함수는 `(0, 0)`를 반환합니다.

가용성: 유닉스.

os.**WCONTINUED**
이 옵션은 자식 프로세스의 상태가 마지막으로 보고된 이후에 작업 제어 중지에서 재개한 경우 보고되도록 합니다.

가용성: 일부 유닉스 시스템.

os.**WUNTRACED**
이 옵션은 자식 프로세스가 중지되었지만, 현재 상태가 중지된 이후 보고되지 않았으면 보고되게 합니다.

가용성: 유닉스.

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.

This function should be employed only if *WIFSIGNALED()* is true.

가용성: 유닉스.

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`.

See *WCONTINUED* option.

가용성: 유닉스.

os.**WIFSTOPPED**(*status*)
Return `True` if the process was stopped by delivery of a signal, otherwise return `False`.

WIFSTOPPED() only returns `True` if the *waitpid()* call was done using *WUNTRACED* option or when the process is being traced (see *ptrace(2)*).

가용성: 유닉스.

os.**WIFSIGNALED**(*status*)
Return `True` if the process was terminated by a signal, otherwise return `False`.

가용성: 유닉스.

os.**WIFEXITED**(*status*)
Return `True` if the process exited terminated normally, that is, by calling `exit()` or `_exit()`, or by returning from `main()`; otherwise return `False`.

가용성: 유닉스.

os.**WEXITSTATUS**(*status*)
Return the process exit status.

This function should be employed only if *WIFEXITED()* is true.

가용성: 유닉스.

os.**WSTOPSIG**(*status*)
프로세스를 멈추게 한 시그널을 반환합니다.

This function should be employed only if *WIFSTOPPED()* is true.

가용성: 유닉스.

os.**WTERMSIG**(*status*)
Return the number of the signal that caused the process to terminate.

This function should be employed only if *WIFSIGNALED()* is true.

가용성: 유닉스.

16.1.7 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.

버전 3.3에 추가.

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_IDLE**
매우 낮은 우선순위의 배경 작업에 대한 스케줄 정책.

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*)
PID가 *pid*인 프로세스의 스케줄 정책을 설정합니다. *pid* 가 0이면, 호출하는 프로세스를 의미합니다. *policy* 는 위의 스케줄 정책 상수 중 하나입니다. *param* 은 *sched_param* 인스턴스입니다.

os.**sched_getscheduler**(*pid*)
PID가 *pid*인 프로세스의 스케줄 정책을 반환합니다. *pid* 가 0이면, 호출하는 프로세스를 의미합니다. 결과는 위의 스케줄 정책 상수 중 하나입니다.

os.**sched_setparam**(*pid*, *param*)
PID가 *pid*인 프로세스의 스케줄 매개 변수를 설정합니다. *pid* 가 0이면 호출하는 프로세스를 의미합니다. *param* 은 *sched_param* 인스턴스입니다.

os.**sched_getparam**(*pid*)
PID가 *pid*인 프로세스의 스케줄 매개 변수를 *sched_param* 인스턴스로 반환합니다. *pid* 가 0이면 호출하는 프로세스를 의미합니다.

os.**sched_rr_get_interval**(*pid*)
PID가 *pid*인 프로세스의 라운드 로빈 퀀텀을 초 단위로 반환합니다. *pid* 가 0이면 호출하는 프로세스를 의미합니다.

os.**sched_yield**()
자발적으로 CPU를 양도합니다.

os.**sched_setaffinity**(*pid*, *mask*)
PID가 *pid*인 프로세스(또는 0이면 현재 프로세스)를 CPU 집합으로 제한합니다. *mask* 는 프로세스가 제한되어야 하는 CPU 집합을 나타내는 정수의 이터러블입니다.

os.**sched_getaffinity**(*pid*)
PID가 *pid*인 프로세스(또는 0이면 현재 프로세스)가 제한되는 CPU 집합을 반환합니다.

16.1.8 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: 유닉스.

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: 유닉스.

os.**cpu_count**()
시스템의 CPU 수를 반환합니다. 파악할 수 없으면, `None`을 반환합니다.

이 숫자는 현재 프로세스에서 사용할 수 있는 CPU 수와 같지 않습니다. 사용 가능한 CPU 수는 `len(os.sched_getaffinity(0))`로 얻을 수 있습니다.

버전 3.4에 추가.

os.**getloadavg**()
마지막 1, 5, 15분에 걸쳐 평균한 시스템 실행 대기열의 프로세스 수를 반환하거나, 로드 평균을 얻을 수 없으면, *OSError*를 발생시킵니다.

가용성: 유닉스.

os.**sysconf**(*name*)
정숫값 시스템 구성 값을 반환합니다. *name* 으로 지정된 구성 값이 정의되어 있지 않으면, `-1`이 반환됩

니다. *confstr()*의 *name* 매개 변수에 관한 주석은 여기에도 적용됩니다; 알려진 이름에 대한 정보를 제공하는 딕셔너리는 sysconf_names에 의해 제공됩니다.

가용성: 유닉스.

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: 유닉스.

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.

버전 3.3에 추가.

16.1.9 난수

os.**getrandom**(*size*, *flags=0*)
최대 *size* 크기의 난수 바이트열을 얻습니다. 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.

flags 인자는 다음 값 중 0개 이상의 값들과 함께 OR 될 수 있는 비트 마스크입니다: *os.GRND_RANDOM* 및 *GRND_NONBLOCK*.

리눅스 getrandom() 매뉴얼 페이지도 참조하십시오.

가용성: 리눅스 3.17 이상.

버전 3.6에 추가.

os.**urandom**(*size*)
암호화에 적합한 *size* 크기의 난수 바이트열을 돌려줍니다.

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 128비트의 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.

윈도우에서, `CryptGenRandom()`을 사용합니다.

더 보기:

secrets 모듈은 고수준 함수를 제공합니다. 플랫폼에서 제공되는 난수 발생기에 대한 사용하기 쉬운 인터페이스는 *random.SystemRandom*를 참조하십시오.

버전 3.6.0에서 변경: 리눅스에서, `getrandom()`은 이제 보안을 강화하기 위해 블로킹 모드로 사용됩니다.

버전 3.5.2에서 변경: 리눅스에서, `getrandom()` 시스템 호출이 블록 하면 (urandom 엔트로피 풀이 아직 초기화되지 않았으면), `/dev/urandom`을 읽는 것으로 대체됩니다.

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.

os.**GRND_NONBLOCK**
기본적으로, `/dev/random`에서 읽을 때, *getrandom()*는 사용할 수 있는 난수 바이트열이 없으면 블록하고, `/dev/urandom`에서 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.

버전 3.6에 추가.

os.**GRND_RANDOM**
이 비트가 설정되면, /dev/urandom 풀 대신 /dev/random 풀에서 난수 바이트열을 얻습니다.

버전 3.6에 추가.

16.2 io — Core tools for working with streams

Source code: Lib/io.py

16.2.1 Overview

The *io* module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O: *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a *file object*. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a *str* object to the `write()` method of a binary stream will raise a *TypeError*. So will giving a *bytes* object to the `write()` method of a text stream.

버전 3.3에서 변경: Operations that used to raise *IOError* now raise *OSError*, since *IOError* is now an alias of *OSError*.

Text I/O

Text I/O expects and produces *str* objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with *open()*, optionally specifying an encoding:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as *StringIO* objects:

```
f = io.StringIO("some initial text data")
```

The text stream API is described in detail in the documentation of *TextIOBase*.

Binary I/O

Binary I/O (also called *buffered I/O*) expects *bytes-like objects* and produces `bytes` objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with `open()` with `'b'` in the mode string:

```
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as `BytesIO` objects:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of `BufferedIOBase`.

Other library modules may provide additional ways to create text or binary streams. See `socket.socket.makefile()` for example.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled:

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of `RawIOBase`.

16.2.2 High-level Module Interface

io.DEFAULT_BUFFER_SIZE
An int containing the default buffer size used by the module's buffered I/O classes. `open()` uses the file's blksize (as obtained by `os.stat()`) if possible.

io.open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)
This is an alias for the builtin `open()` function.

exception io.BlockingIOError
This is a compatibility alias for the builtin `BlockingIOError` exception.

exception io.UnsupportedOperation
An exception inheriting `OSError` and `ValueError` that is raised when an unsupported operation is called on a stream.

In-memory streams

It is also possible to use a *str* or *bytes-like object* as a file for both reading and writing. For strings *StringIO* can be used like a file opened in text mode. *BytesIO* can be used like a file opened in binary mode. Both provide full read-write capabilities with random access.

더 보기:

sys contains the standard IO streams: *sys.stdin*, *sys.stdout*, and *sys.stderr*.

16.2.3 Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First *abstract base classes* (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

> 참고: The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, *BufferedIOBase* provides unoptimized implementations of `readinto()` and *readline()*.

At the top of the I/O hierarchy is the abstract base class *IOBase*. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise *UnsupportedOperation* if they do not support a given operation.

The *RawIOBase* ABC extends *IOBase*. It deals with the reading and writing of bytes to a stream. *FileIO* subclasses *RawIOBase* to provide an interface to files in the machine's file system.

The *BufferedIOBase* ABC deals with buffering on a raw byte stream (*RawIOBase*). Its subclasses, *BufferedWriter*, *BufferedReader*, and *BufferedRWPair* buffer streams that are readable, writable, and both readable and writable. *BufferedRandom* provides a buffered interface to random access streams. Another *BufferedIOBase* subclass, *BytesIO*, is a stream of in-memory bytes.

The *TextIOBase* ABC, another subclass of *IOBase*, deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. *TextIOWrapper*, which extends it, is a buffered text interface to a buffered raw stream (*BufferedIOBase*). Finally, *StringIO* is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of *open()* are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the *io* module:

ABC	Inherits	Stub Methods	Mixin Methods and Properties
IOBase		`fileno`, `seek`, and `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable`, and `writelines`
RawIOBase	*IOBase*	`readinto` and `write`	Inherited *IOBase* methods, `read`, and `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1`, and `write`	Inherited *IOBase* methods, `readinto`, and `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline`, and `write`	Inherited *IOBase* methods, `encoding`, `errors`, and `newlines`

I/O Base Classes

class io.**IOBase**

The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though *IOBase* does not declare `read()` or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a *ValueError* (or *UnsupportedOperation*) when operations they do not support are called.

The basic type used for binary data read from or written to a file is *bytes*. Other *bytes-like objects* are accepted as method arguments too. Text I/O classes work with *str* data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise *ValueError* in this case.

IOBase (and its subclasses) supports the iterator protocol, meaning that an *IOBase* object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See *readline()* below.

IOBase is also a context manager and therefore supports the `with` statement. In this example, *file* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase provides these data attributes and methods:

close()

Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a *ValueError*.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

closed

`True` if the stream is closed.

fileno()
Return the underlying file descriptor (an integer) of the stream if it exists. An *OSError* is raised if the IO object does not use a file descriptor.

flush()
Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()
Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

readable()
Return `True` if the stream can be read from. If `False`, `read()` will raise *OSError*.

readline(*size=-1*)
Read and return one line from the stream. If *size* is specified, at most *size* bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the *newline* argument to *open()* can be used to select the line terminator(s) recognized.

readlines(*hint=-1*)
Read and return a list of lines from the stream. *hint* can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds *hint*.

Note that it's already possible to iterate on file objects using `for line in file: ...` without calling `file.readlines()`.

seek(*offset*, *whence=SEEK_SET*)
Change the stream position to the given byte *offset*. *offset* is interpreted relative to the position indicated by *whence*. The default value for *whence* is `SEEK_SET`. Values for *whence* are:

- `SEEK_SET` or `0` – start of the stream (the default); *offset* should be zero or positive
- `SEEK_CUR` or `1` – current stream position; *offset* may be negative
- `SEEK_END` or `2` – end of the stream; *offset* is usually negative

Return the new absolute position.

버전 3.1에 추가: The `SEEK_*` constants.

버전 3.3에 추가: Some operating systems could support additional values, like `os.SEEK_HOLE` or `os.SEEK_DATA`. The valid values for a file could depend on it being open in text or binary mode.

seekable()
Return `True` if the stream supports random access. If `False`, *seek()*, *tell()* and *truncate()* will raise *OSError*.

tell()
Return the current stream position.

truncate(*size=None*)
Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

버전 3.5에서 변경: Windows will now zero-fill files when extending.

writable()
Return `True` if the stream supports writing. If `False`, `write()` and *truncate()* will raise *OSError*.

writelines(*lines*)
Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()
Prepare for object destruction. *IOBase* provides a default implementation of this method that calls the instance's *close()* method.

class io.**RawIOBase**
Base class for raw binary I/O. It inherits *IOBase*. There is no public constructor.

Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from *IOBase*, *RawIOBase* provides the following methods:

read(*size=-1*)
Read up to *size* bytes from the object and return them. As a convenience, if *size* is unspecified or -1, all bytes until EOF are returned. Otherwise, only one system call is ever made. Fewer than *size* bytes may be returned if the operating system call returns fewer than *size* bytes.

If 0 bytes are returned, and *size* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

The default implementation defers to *readall()* and *readinto()*.

readall()
Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(*b*)
Read bytes into a pre-allocated, writable *bytes-like object* *b*, and return the number of bytes read. For example, *b* might be a *bytearray*. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*)
Write the given *bytes-like object*, *b*, to the underlying raw stream, and return the number of bytes written. This can be less than the length of *b* in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. `None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class io.**BufferedIOBase**
Base class for binary streams that support some kind of buffering. It inherits *IOBase*. There is no public constructor.

The main difference with *RawIOBase* is that methods *read()*, *readinto()* and *write()* will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise *BlockingIOError* if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their *RawIOBase* counterparts, they will never return `None`.

Besides, the *read()* method does not have a default implementation that defers to *readinto()*.

A typical *BufferedIOBase* implementation should not inherit from a *RawIOBase* implementation, but wrap one, like *BufferedWriter* and *BufferedReader* do.

BufferedIOBase provides or overrides these methods and attribute in addition to those from *IOBase*:

raw
The underlying raw stream (a *RawIOBase* instance) that *BufferedIOBase* deals with. This is not part of the *BufferedIOBase* API and may not exist on some implementations.

detach()
Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like *BytesIO*, do not have the concept of a single raw stream to return from this method. They raise *UnsupportedOperation*.

버전 3.1에 추가.

read(*size=-1*)

Read and return up to *size* bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty *bytes* object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A *BlockingIOError* is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

read1([*size*])

Read and return up to *size* bytes, with at most one call to the underlying raw stream's *read()* (or *readinto()*) method. This can be useful if you are implementing your own buffering on top of a *BufferedIOBase* object.

If *size* is `-1` (the default), an arbitrary number of bytes are returned (more than zero unless EOF is reached).

readinto(*b*)

Read bytes into a pre-allocated, writable *bytes-like object* *b* and return the number of bytes read. For example, *b* might be a *bytearray*.

Like *read()*, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.

A *BlockingIOError* is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

readinto1(*b*)

Read bytes into a pre-allocated, writable *bytes-like object* *b*, using at most one call to the underlying raw stream's *read()* (or *readinto()*) method. Return the number of bytes read.

A *BlockingIOError* is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

버전 3.5에 추가.

write(*b*)

Write the given *bytes-like object*, *b*, and return the number of bytes written (always equal to the length of *b* in bytes, since if the write fails an *OSError* will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a *BlockingIOError* is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

Raw File I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

FileIO represents an OS-level file containing bytes data. It implements the *RawIOBase* interface (and therefore the *IOBase* interface, too).

The *name* can be one of two things:

- a character string or *bytes* object representing the path to the file which will be opened. In this case closefd must be `True` (the default) otherwise an error will be raised.
- an integer representing the number of an existing OS-level file descriptor to which the resulting *FileIO* object will give access. When the FileIO object is closed this fd will be closed as well, unless *closefd* is set to `False`.

The *mode* can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. *FileExistsError* will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The `read()` (when called with a positive argument), `readinto()` and `write()` methods on this class will only make one system call.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*name*, *flags*). *opener* must return an open file descriptor (passing *os.open* as *opener* results in functionality similar to passing `None`).

The newly created file is *non-inheritable*.

See the *open()* built-in function for examples on using the *opener* parameter.

버전 3.3에서 변경: The *opener* parameter was added. The `'x'` mode was added.

버전 3.4에서 변경: The file is now non-inheritable.

In addition to the attributes and methods from *IOBase* and *RawIOBase*, *FileIO* provides the following data attributes:

mode
The mode as given in the constructor.

name
The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.**BytesIO**([*initial_bytes*])

A stream implementation using an in-memory bytes buffer. It inherits *BufferedIOBase*. The buffer is discarded when the *close()* method is called.

The optional argument *initial_bytes* is a *bytes-like object* that contains initial data.

BytesIO provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

getbuffer()
Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

참고: As long as the view exists, the *BytesIO* object cannot be resized or closed.

버전 3.2에 추가.

getvalue()
Return *bytes* containing the entire contents of the buffer.

read1([*size*])
In *BytesIO*, this is the same as *read()*.

버전 3.7에서 변경: The *size* argument is now optional.

readinto1(*b*)
In *BytesIO*, this is the same as *readinto()*.

버전 3.5에 추가.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffer providing higher-level access to a readable, sequential *RawIOBase* object. It inherits *BufferedIOBase*. When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a *BufferedReader* for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, *DEFAULT_BUFFER_SIZE* is used.

BufferedReader provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

peek([*size*])
Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read([*size*])
Read and return *size* bytes, or if *size* is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1([*size*])
Read and return up to *size* bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

버전 3.7에서 변경: The *size* argument is now optional.

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffer providing higher-level access to a writeable, sequential *RawIOBase* object. It inherits *BufferedIOBase*. When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying *RawIOBase* object under various conditions, including:

- when the buffer gets too small for all pending data;
- when *flush()* is called;
- when a seek() is requested (for *BufferedRandom* objects);
- when the *BufferedWriter* object is closed or destroyed.

The constructor creates a *BufferedWriter* for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedWriter provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

flush()
Force bytes held in the buffer into the raw stream. A *BlockingIOError* should be raised if the raw stream blocks.

write(*b*)
Write the *bytes-like object*, *b*, and return the number of bytes written. When in non-blocking mode, a *BlockingIOError* is raised if the buffer needs to be written out but the raw stream blocks.

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffered interface to random access streams. It inherits *BufferedReader* and *BufferedWriter*, and further supports `seek()` and `tell()` functionality.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the *buffer_size* is omitted it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedRandom is capable of anything *BufferedReader* or *BufferedWriter* can do.

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffered I/O object combining two unidirectional *RawIOBase* objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits *BufferedIOBase*.

reader and *writer* are *RawIOBase* objects that are readable and writeable respectively. If the *buffer_size* is omitted it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedRWPair implements all of *BufferedIOBase*'s methods except for *detach()*, which raises *UnsupportedOperation*.

> **경고:** *BufferedRWPair* does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use *BufferedRandom* instead.

Text I/O

class io.**TextIOBase**
Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits *IOBase*. There is no public constructor.

TextIOBase provides or overrides these data attributes and methods in addition to those from *IOBase*:

encoding
The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

errors
The error setting of the decoder or encoder.

newlines
A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer
The underlying binary buffer (a *BufferedIOBase* instance) that *TextIOBase* deals with. This is not part of the *TextIOBase* API and may not exist in some implementations.

detach()
Separate the underlying binary buffer from the *TextIOBase* and return it.

After the underlying buffer has been detached, the *TextIOBase* is in an unusable state.

Some *TextIOBase* implementations, like *StringIO*, may not have the concept of an underlying buffer and calling this method will raise *UnsupportedOperation*.

버전 3.1에 추가.

read(*size=-1*)
Read and return at most *size* characters from the stream as a single *str*. If *size* is negative or None, reads until EOF.

readline(*size=-1*)
Read until newline or EOF and return a single str. If the stream is already at EOF, an empty string is returned.

If *size* is specified, at most *size* characters will be read.

seek(*offset*, *whence=SEEK_SET*)
Change the stream position to the given *offset*. Behaviour depends on the *whence* parameter. The default value for *whence* is SEEK_SET.

- SEEK_SET or 0: seek from the start of the stream (the default); *offset* must either be a number returned by *TextIOBase.tell()*, or zero. Any other *offset* value produces undefined behaviour.
- SEEK_CUR or 1: "seek" to the current position; *offset* must be zero, which is a no-operation (all other values are unsupported).
- SEEK_END or 2: seek to the end of the stream; *offset* must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

버전 3.1에 추가: The SEEK_* constants.

tell()
Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(*s*)
Write the string *s* to the stream and return the number of characters written.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)
A buffered text stream over a *BufferedIOBase* binary stream. It inherits *TextIOBase*.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to *locale.getpreferredencoding(False)*.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass 'strict' to raise a *ValueError* exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) 'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data. 'backslashreplace' causes malformed data to be replaced by a backslashed escape sequence. When writing, 'xmlcharrefreplace' (replace with the appropriate XML character reference) or 'namereplace' (replace with \N{...} escape sequences) can be used. Any other error handling name that has been registered with *codecs.register_error()* is also valid.

newline controls how line endings are handled. It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

- When reading input from the stream, if *newline* is None, *universal newlines* mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the

caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

- When writing output to the stream, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, *os.linesep*. If *newline* is `''` or `'\n'`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *line_buffering* is `True`, `flush()` is implied when a call to write contains a newline character or a carriage return.

If *write_through* is `True`, calls to `write()` are guaranteed not to be buffered: any data written on the *TextIOWrapper* object is immediately handled to its underlying binary *buffer*.

버전 3.3에서 변경: The *write_through* argument has been added.

버전 3.3에서 변경: The default *encoding* is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don't change temporary the locale encoding using *locale.setlocale()*, use the current locale encoding instead of the user preferred encoding.

TextIOWrapper provides these members in addition to those of *TextIOBase* and its parents:

line_buffering
Whether line buffering is enabled.

write_through
Whether writes are passed immediately to the underlying binary buffer.

버전 3.7에 추가.

reconfigure(**, encoding][, errors][, newline][, line_buffering][, write_through]*)
Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except `errors='strict'` is used when *encoding* is specified but *errors* is not specified.

It is not possible to change the encoding or newline if some data has already been read from the stream. On the other hand, changing encoding after write is possible.

This method does an implicit stream flush before setting the new parameters.

버전 3.7에 추가.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)
An in-memory stream for text I/O. The text buffer is discarded when the *close()* method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by *write()*. The stream is positioned at the start of the buffer.

The *newline* argument works like that of *TextIOWrapper*. The default is to consider only `\n` characters as ends of lines and to do no newline translation. If *newline* is set to `None`, newlines are written as `\n` on all platforms, but universal newline decoding is still performed when reading.

StringIO provides this method in addition to those from *TextIOBase* and its parents:

getvalue()
Return a `str` containing the entire contents of the buffer. Newlines are decoded as if by *read()*, although the stream position is not changed.

Example usage:

```
import io
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**

A helper codec that decodes newlines for *universal newlines* mode. It inherits *codecs.IncrementalDecoder*.

16.2.4 Performance

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, `TextIOWrapper.tell()` and `TextIOWrapper.seek()` are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to *BytesIO*.

Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a *signal* handler. If a thread tries to re-enter a buffered object which it is already accessing, a *RuntimeError* is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the *open()* function will wrap a buffered object inside a *TextIOWrapper*. This includes standard streams and therefore affects the built-in function *print()* as well.

16.3 `time` — Time access and conversions

This module provides various time-related functions. For related functionality, see also the *datetime* and *calendar* modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The *epoch* is the point where the time starts, and is platform dependent. For Unix, the epoch is January 1, 1970, 00:00:00 (UTC). To find out what the epoch is on a given platform, look at `time.gmtime(0)`.
- The term *seconds since the epoch* refers to the total number of elapsed seconds since the epoch, typically excluding leap seconds. Leap seconds are excluded from this total on all POSIX-compliant platforms.
- The functions in this module may not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.
- Function *strptime()* can parse 2-digit years when given `%y` format code. When 2-digit years are parsed, they are converted according to the POSIX and ISO C standards: values 69–99 are mapped to 1969–1999, and values 0–68 are mapped to 2000–2068.
- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.
- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.
- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock "ticks" only 50 or 100 times a second.
- On the other hand, the precision of *time()* and *sleep()* is better than their Unix equivalents: times are expressed as floating point numbers, *time()* returns the most accurate time available (using Unix `gettimeofday()` where available), and *sleep()* will accept a time with a nonzero fraction (Unix `select()` is used to implement this, where available).
- The time value as returned by *gmtime()*, *localtime()*, and *strptime()*, and accepted by *asctime()*, *mktime()* and *strftime()*, is a sequence of 9 integers. The return values of *gmtime()*, *localtime()*, and *strptime()* also offer attribute names for individual fields.

  See *struct_time* for a description of these objects.

버전 3.3에서 변경: The *struct_time* type was extended to provide the `tm_gmtoff` and `tm_zone` attributes when platform supports corresponding `struct tm` members.

버전 3.6에서 변경: The *struct_time* attributes `tm_gmtoff` and `tm_zone` are now available on all platforms.

- Use the following functions to convert between time representations:

From	To	Use
seconds since the epoch	*struct_time* in UTC	*gmtime()*
seconds since the epoch	*struct_time* in local time	*localtime()*
struct_time in UTC	seconds since the epoch	*calendar.timegm()*
struct_time in local time	seconds since the epoch	*mktime()*

16.3.1 Functions

`time.`**`asctime`**`([t])`
Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string of the following form: `'Sun Jun 20 23:21:05 1993'`. If *t* is not provided, the current time as returned by *localtime()* is used. Locale information is not used by *asctime()*.

참고: Unlike the C function of the same name, *asctime()* does not add a trailing newline.

`time.`**`clock`**`()`
On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of "processor time", depends on that of the C function of the same name.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function `QueryPerformanceCounter()`. The resolution is typically better than one microsecond.

Deprecated since version 3.3, will be removed in version 3.8: The behaviour of this function depends on the platform: use *perf_counter()* or *process_time()* instead, depending on your requirements, to have a well defined behaviour.

`time.`**`pthread_getcpuclockid`**`(thread_id)`
Return the *clk_id* of the thread-specific CPU-time clock for the specified *thread_id*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

경고: Passing an invalid or expired *thread_id* may result in undefined behavior, such as segmentation fault.

Availability: Unix (see the man page for *pthread_getcpuclockid(3)* for further information).

버전 3.7에 추가.

`time.`**`clock_getres`**`(clk_id)`
Return the resolution (precision) of the specified clock *clk_id*. Refer to *Clock ID Constants* for a list of accepted values for *clk_id*.

Availability: Unix.

버전 3.3에 추가.

time.**clock_gettime**(*clk_id*) → float
Return the time of the specified clock *clk_id*. Refer to *Clock ID Constants* for a list of accepted values for *clk_id*.

Availability: Unix.

버전 3.3에 추가.

time.**clock_gettime_ns**(*clk_id*) → int
Similar to `clock_gettime()` but return time as nanoseconds.

Availability: Unix.

버전 3.7에 추가.

time.**clock_settime**(*clk_id*, *time: float*)
Set the time of the specified clock *clk_id*. Currently, `CLOCK_REALTIME` is the only accepted value for *clk_id*.

Availability: Unix.

버전 3.3에 추가.

time.**clock_settime_ns**(*clk_id*, *time: int*)
Similar to `clock_settime()` but set time with nanoseconds.

Availability: Unix.

버전 3.7에 추가.

time.**ctime**([*secs*])
Convert a time expressed in seconds since the epoch to a string representing local time. If *secs* is not provided or `None`, the current time as returned by `time()` is used. `ctime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by `ctime()`.

time.**get_clock_info**(*name*)
Get information on the specified clock as a namespace object. Supported clock names and the corresponding functions to read their value are:

- `'clock'`: `time.clock()`
- `'monotonic'`: `time.monotonic()`
- `'perf_counter'`: `time.perf_counter()`
- `'process_time'`: `time.process_time()`
- `'thread_time'`: `time.thread_time()`
- `'time'`: `time.time()`

The result has the following attributes:

- *adjustable*: `True` if the clock can be changed automatically (e.g. by a NTP daemon) or manually by the system administrator, `False` otherwise
- *implementation*: The name of the underlying C function used to get the clock value. Refer to *Clock ID Constants* for possible values.
- *monotonic*: `True` if the clock cannot go backward, `False` otherwise
- *resolution*: The resolution of the clock in seconds (`float`)

버전 3.3에 추가.

time.**gmtime**([*secs*])
Convert a time expressed in seconds since the epoch to a `struct_time` in UTC in which the dst flag is always zero. If *secs* is not provided or `None`, the current time as returned by `time()` is used. Fractions of a second are

ignored. See above for a description of the *struct_time* object. See *calendar.timegm()* for the inverse of this function.

time.**localtime**([*secs*])
Like *gmtime()* but converts to local time. If *secs* is not provided or *None*, the current time as returned by *time()* is used. The dst flag is set to `1` when DST applies to the given time.

time.**mktime**(*t*)
This is the inverse function of *localtime()*. Its argument is the *struct_time* or full 9-tuple (since the dst flag is needed; use `-1` as the dst flag if it is unknown) which expresses the time in *local* time, not UTC. It returns a floating point number, for compatibility with *time()*. If the input value cannot be represented as a valid time, either *OverflowError* or *ValueError* will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

time.**monotonic**() → float
Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

버전 3.3에 추가.

버전 3.5에서 변경: The function is now always available and always system-wide.

time.**monotonic_ns**() → int
Similar to *monotonic()*, but return time as nanoseconds.

버전 3.7에 추가.

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 and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

버전 3.3에 추가.

time.**perf_counter_ns**() → int
Similar to *perf_counter()*, but return time as nanoseconds.

버전 3.7에 추가.

time.**process_time**() → float
Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

버전 3.3에 추가.

time.**process_time_ns**() → int
Similar to *process_time()* but return time as nanoseconds.

버전 3.7에 추가.

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. The actual suspension time may be less than that requested because any caught signal will terminate the *sleep()* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

버전 3.5에서 변경: The function now sleeps at least *secs* even if the sleep is interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale).

time.**strftime**(*format*[, *t*])

Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string as specified by the *format* argument. If *t* is not provided, the current time as returned by *localtime()* is used. *format* must be a string. *ValueError* is raised if any field in *t* is outside of the allowed range.

0 is a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

The following directives can be embedded in the *format* string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the *strftime()* result:

Directive	Meaning	Notes
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	(1)
%S	Second as a decimal number [00,61].	(2)
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(3)
%w	Weekday as a decimal number [0(Sunday),6].	
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(3)
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].	
%Z	Time zone name (no characters if no time zone exists).	
%%	A literal '%' character.	

Notes:

(1) When used with the *strptime()* function, the %p directive only affects the output hour field if the %I directive is used to parse the hour.

(2) The range really is 0 to 61; value 60 is valid in timestamps representing leap seconds and value 61 is supported for historical reasons.

(3) When used with the *strptime()* function, %U and %W are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the **RFC 2822** Internet email standard.[1]

[1] The use of %Z is now deprecated, but the %z escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 **RFC 822** standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation.

On some platforms, an optional field width and precision specification can immediately follow the initial `'%'` of a directive in the following order; this is also not portable. The field width is normally 2 except for `%j` where it is 3.

`time.`**`strptime`**(*string*[, *format*])

Parse a string representing a time according to a format. The return value is a *struct_time* as returned by *gmtime()* or *localtime()*.

The *format* parameter uses the same directives as those used by *strftime()*; it defaults to `"%a %b %d %H:%M:%S %Y"` which matches the formatting returned by *ctime()*. If *string* cannot be parsed according to *format*, or if it has excess data after parsing, *ValueError* is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are `(1900, 1, 1, 0, 0, 0, 0, 1, -1)`. Both *string* and *format* must be strings.

For example:

```
>>> 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)
```

Support for the `%Z` directive is based on the values contained in `tzname` and whether `daylight` is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because `strftime()` is implemented per platform it can sometimes offer more directives than those listed. But `strptime()` is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

`class` `time.`**`struct_time`**

The type of the time value sequence returned by *gmtime()*, *localtime()*, and *strptime()*. It is an object with a *named tuple* interface: values can be accessed by index and by attribute name. The following values are present:

Index	Attribute	Values
0	`tm_year`	(for example, 1993)
1	`tm_mon`	range [1, 12]
2	`tm_mday`	range [1, 31]
3	`tm_hour`	range [0, 23]
4	`tm_min`	range [0, 59]
5	`tm_sec`	range [0, 61]; see **(2)** in *strftime()* description
6	`tm_wday`	range [0, 6], Monday is 0
7	`tm_yday`	range [1, 366]
8	`tm_isdst`	0, 1 or -1; see below
N/A	`tm_zone`	abbreviation of timezone name
N/A	`tm_gmtoff`	offset east of UTC in seconds

before the year 2000. After that, **RFC 822** became obsolete and the 4-digit year has been first recommended by **RFC 1123** and then mandated by **RFC 2822**.

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11].

In calls to *mktime()*, tm_isdst may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

When a tuple with an incorrect length is passed to a function expecting a *struct_time*, or having elements of the wrong type, a *TypeError* is raised.

time.**time**() → float
Return the time in seconds since the *epoch* as a floating point number. The specific date of the epoch and the handling of leap seconds is platform dependent. On Windows and most Unix systems, the epoch is January 1, 1970, 00:00:00 (UTC) and leap seconds are not counted towards the time in seconds since the epoch. This is commonly referred to as Unix time. To find out what the epoch is on a given platform, look at gmtime(0).

Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

The number returned by *time()* may be converted into a more common time format (i.e. year, month, day, hour, etc…) in UTC by passing it to *gmtime()* function or in local time by passing it to the *localtime()* function. In both cases a *struct_time* object is returned, from which the components of the calendar date may be accessed as attributes.

time.**thread_time**() → float
Return the value (in fractional seconds) of the sum of the system and user CPU time of the current thread. It does not include time elapsed during sleep. It is thread-specific by definition. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls in the same thread is valid.

Availability: Windows, Linux, Unix systems supporting CLOCK_THREAD_CPUTIME_ID.

버전 3.7에 추가.

time.**thread_time_ns**() → int
Similar to *thread_time()* but return time as nanoseconds.

버전 3.7에 추가.

time.**time_ns**() → int
Similar to *time()* but returns time as an integer number of nanoseconds since the *epoch*.

버전 3.7에 추가.

time.**tzset**()
Reset the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. It will also set the variables tzname (from the TZ environment variable), timezone (non-DST seconds West of UTC), altzone (DST seconds west of UTC) and daylight (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

Availability: Unix.

참고: Although in many cases, changing the TZ environment variable may affect the output of functions like *localtime()* without calling *tzset()*, this behavior should not be relied on.

The TZ environment variable should contain no whitespace.

The standard format of the TZ environment variable is (whitespace added for clarity):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Where the components are:

std and dst Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname

offset The offset has the form: `± hh[:mm[:ss]]`. This indicates the value added the local time to arrive at UTC. If preceded by a '-', the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

start[/time], end[/time] Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

Jn The Julian day *n* (1 <= *n* <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n The zero-based Julian day (0 <= *n* <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d The *d*'th day (0 <= *d* <= 6) of week *n* of month *m* of the year (1 <= *n* <= 5, 1 <= *m* <= 12, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is a Sunday.

`time` has the same format as `offset` except that no leading sign ('-' or '+') is allowed. The default, if time is not given, is 02:00:00.

```
>>> 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'
```

On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system's zoneinfo (*tzfile(5)*) database to specify the timezone rules. To do this, set the `TZ` environment variable to the path of the required timezone datafile, relative to the root of the systems 'zoneinfo' timezone database, usually located at `/usr/share/zoneinfo`. For example, `'US/Eastern'`, `'Australia/Melbourne'`, `'Egypt'` or `'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 Clock ID Constants

These constants are used as parameters for *clock_getres()* and *clock_gettime()*.

`time.`**CLOCK_BOOTTIME**
Identical to *CLOCK_MONOTONIC*, except it also includes any time that the system is suspended.

This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of *CLOCK_REALTIME*, which may have discontinuities if the time is changed using `settimeofday()` or similar.

Availability: Linux 2.6.39 or later.

버전 3.7에 추가.

time.**CLOCK_HIGHRES**
The Solaris OS has a CLOCK_HIGHRES timer that attempts to use an optimal hardware source, and may give close to nanosecond resolution. CLOCK_HIGHRES is the nonadjustable, high-resolution clock.

Availability: Solaris.

버전 3.3에 추가.

time.**CLOCK_MONOTONIC**
Clock that cannot be set and represents monotonic time since some unspecified starting point.

Availability: Unix.

버전 3.3에 추가.

time.**CLOCK_MONOTONIC_RAW**
Similar to *CLOCK_MONOTONIC*, but provides access to a raw hardware-based time that is not subject to NTP adjustments.

Availability: Linux 2.6.28 and newer, macOS 10.12 and newer.

버전 3.3에 추가.

time.**CLOCK_PROCESS_CPUTIME_ID**
High-resolution per-process timer from the CPU.

Availability: Unix.

버전 3.3에 추가.

time.**CLOCK_PROF**
High-resolution per-process timer from the CPU.

Availability: FreeBSD, NetBSD 7 or later, OpenBSD.

버전 3.7에 추가.

time.**CLOCK_THREAD_CPUTIME_ID**
Thread-specific CPU-time clock.

Availability: Unix.

버전 3.3에 추가.

time.**CLOCK_UPTIME**
Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement, both absolute and interval.

Availability: FreeBSD, OpenBSD 5.5 or later.

버전 3.7에 추가.

The following constant is the only parameter that can be sent to *clock_settime()*.

time.**CLOCK_REALTIME**
System-wide real-time clock. Setting this clock requires appropriate privileges.

Availability: Unix.

버전 3.3에 추가.

16.3.3 Timezone Constants

`time.`**`altzone`**
The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if `daylight` is nonzero. See note below.

`time.`**`daylight`**
Nonzero if a DST timezone is defined. See note below.

`time.`**`timezone`**
The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK). See note below.

`time.`**`tzname`**
A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used. See note below.

참고: For the above Timezone constants (*`altzone`*, *`daylight`*, *`timezone`*, and *`tzname`*), the value is determined by the timezone rules in effect at module load time or the last time *`tzset()`* is called and may be incorrect for times in the past. It is recommended to use the `tm_gmtoff` and `tm_zone` results from *`localtime()`* to obtain timezone information.

더 보기:

Module *`datetime`* More object-oriented interface to dates and times.

Module *`locale`* Internationalization services. The locale setting affects the interpretation of many format specifiers in *`strftime()`* and *`strptime()`*.

Module *`calendar`* General calendar-related functions. *`timegm()`* is the inverse of *`gmtime()`* from this module.

16.4 `argparse` — 명령행 옵션, 인자와 부속 명령을 위한 파서

버전 3.2에 추가.

소스 코드: Lib/argparse.py

자습서

이 페이지는 API 레퍼런스 정보를 담고 있습니다. 파이썬 명령행 파싱에 대한 더 친절한 소개를 원하시면, argparse 자습서 를 보십시오.

`argparse` 모듈은 사용자 친화적인 명령행 인터페이스를 쉽게 작성하도록 합니다. 프로그램이 필요한 인자를 정의하면, *`argparse`* 는 *`sys.argv`* 를 어떻게 파싱할지 파악합니다. 또한 *`argparse`* 모듈은 도움말과 사용법 메시지를 자동 생성하고, 사용자가 프로그램에 잘못된 인자를 줄 때 에러를 발생시킵니다.

16.4.1 예

다음 코드는 정수 목록을 받아 합계 또는 최댓값을 출력하는 파이썬 프로그램입니다:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

위의 파이썬 코드가 `prog.py` 라는 파일에 저장되었다고 가정할 때, 명령행에서 실행되고 유용한 도움말 메시지를 제공할 수 있습니다:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

적절한 인자로 실행하면 명령행 정수의 합계 또는 최댓값을 인쇄합니다.:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

잘못된 인자가 전달되면 에러가 발생합니다:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

다음 절에서 이 예제를 자세히 살펴봅니다.

파서 만들기

*argparse*를 사용하는 첫 번째 단계는 *ArgumentParser* 객체를 생성하는 것입니다:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

ArgumentParser 객체는 명령행을 파이썬 데이터형으로 파싱하는데 필요한 모든 정보를 담고 있습니다.

인자 추가하기

ArgumentParser 에 프로그램 인자에 대한 정보를 채우려면 *add_argument()* 메서드를 호출하면 됩니다. 일반적으로 이 호출은 *ArgumentParser* 에게 명령행의 문자열을 객체로 변환하는 방법을 알려줍니다. 이 정보는 저장되고, *parse_args()* 가 호출될 때 사용됩니다. 예를 들면:

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

나중에, *parse_args()* 를 호출하면 두 가지 어트리뷰트, `integers` 와 `accumulate` 를 가진 객체를 반환합니다. `integers` 어트리뷰트는 하나 이상의 int로 구성된 리스트가 될 것이고, `accumulate` 어트리뷰트는 명령행에 `--sum` 가 지정되었을 경우 *sum()* 함수가 되고, 그렇지 않으면 *max()* 함수가 될 것입니다.

인자 파싱하기

ArgumentParser 는 *parse_args()* 메서드를 통해 인자를 파싱합니다. 이 메서드는 명령행을 검사하고 각 인자를 적절한 형으로 변환 한 다음 적절한 액션을 호출합니다. 대부분은, 이것은 간단한 *Namespace* 객체가 명령행에서 파싱 된 어트리뷰트들로 만들어진다는 것을 뜻합니다:

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

스크립트에서, *parse_args()* 는 일반적으로 인자 없이 호출되고, *ArgumentParser* 는 *sys.argv* 에서 자동으로 명령행 인자를 결정합니다.

16.4.2 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*)

새로운 *ArgumentParser* 객체를 만듭니다. 모든 매개 변수는 키워드 인자로 전달되어야 합니다. 매개 변수마다 아래에서 더 자세히 설명되지만, 요약하면 다음과 같습니다:

- *prog* - 프로그램의 이름 (기본값: `sys.argv[0]`)
- *usage* - 프로그램 사용법을 설명하는 문자열 (기본값: 파서에 추가된 인자로부터 만들어지는 값)
- *description* - 인자 도움말 전에 표시할 텍스트 (기본값: none)
- *epilog* - 인자 도움말 후에 표시할 텍스트 (기본값: none)
- *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* - 약어가 모호하지 않으면 긴 옵션을 축약할 수 있도록 합니다. (기본값: True)

버전 3.5에서 변경: *allow_abbrev* 매개 변수가 추가되었습니다.

다음 절에서는 이들 각각의 사용 방법에 관해 설명합니다.

prog

기본적으로, *ArgumentParser* 객체는 sys.argv[0] 을 사용하여 도움말 메시지에 프로그램의 이름을 표시하는 방법을 결정합니다. 이 기본값은 명령행에서 프로그램이 호출된 방법과 도움말 메시지를 일치시키기 때문에 거의 항상 바람직합니다. 예를 들어, 다음 코드가 들어있는 myprogram.py 라는 파일을 생각해보십시오:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

이 프로그램의 도움말은 (프로그램이 어디에서 호출되었는지에 관계없이) 프로그램 이름으로 myprogram.py 를 표시합니다:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

이 기본 동작을 변경하려면, prog= 인자를 *ArgumentParser* 에 사용하여 다른 값을 제공 할 수 있습니다:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

프로그램 이름은 %(prog)s 포맷 지정자를 사용해서 도움말에 쓸 수 있습니다. sys.argv[0] 나 prog= 인자 중 어떤 것으로부터 결정되든 상관없습니다.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

기본적으로, *ArgumentParser* 는 포함 된 인자로부터 사용법 메시지를 계산합니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

기본 메시지는 usage= 키워드 인자로 재정의될 수 있습니다:

```
>>> 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:
 bar          bar help

optional arguments:
 -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.

description

ArgumentParser 생성자에 대한 대부분의 호출은 description= 키워드 인자를 사용할 것입니다. 이 인자는 프로그램의 기능과 작동 방식에 대한 간략한 설명을 제공합니다. 도움말 메시지에서, 설명은 명령행 사용 문자열과 다양한 인자에 대한 도움말 메시지 사이에 표시됩니다:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
optional arguments:
 -h, --help  show this help message and exit
```

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

optional arguments:
 -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])
>>> 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

optional arguments:
 -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
...             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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter 는 인자 설명을 포함하여 모든 종류의 도움말 텍스트에 있는 공백을 유지합니다. 그러나 여러 개의 줄 넘김은 하나로 치환됩니다. 여러 개의 빈 줄을 유지하려면, 줄 바꿈 사이에 스페이스를 추가하십시오.

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 [bar ...]]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
 -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

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

대부분의 명령행 옵션은 `-f/--foo` 처럼 `-` 를 접두어로 사용합니다. `+f` 나 `/foo` 같은 옵션과 같이, 다른 접두어 문자를 지원해야 하는 파서는 ArgumentParser 생성자에 `prefix_chars=` 인자를 사용하여 지정할 수 있습니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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

때로는, 예를 들어 특히 긴 인자 목록을 다룰 때, 인자 목록을 명령행에 입력하는 대신 파일에 보관하는 것이 좋습니다. fromfile_prefix_chars= 인자가 *ArgumentParser* 생성자에 주어지면, 지정된 문자로 시작하는 인자는 파일로 간주하고 파일에 포함된 인자로 대체됩니다. 예를 들면:

```
>>> with open('args.txt', 'w') 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')
```

파일에서 읽은 인자는 기본적으로 한 줄에 하나씩 있어야 하고 (하지만 *convert_arg_line_to_args()* 도 참조하십시오), 명령행에서 원래 파일을 참조하는 인자와 같은 위치에 있는 것처럼 처리됩니다. 위의 예에서 표현식 ['-f', 'foo', '@args.txt'] 는 ['-f', 'foo', '-f', 'bar'] 와 동등하게 취급됩니다.

fromfile_prefix_chars= 인자의 기본값은 None 입니다. 이것은 인자가 절대로 파일 참조로 취급되지 않는다는 것을 의미합니다.

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')
>>> 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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

버전 3.5에 추가.

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]

optional arguments:
 -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

기본적으로, ArgumentParser 객체는 파서의 도움말 메시지를 표시하는 옵션을 추가합니다. 예를 들어, 다음 코드를 포함하는 myprogram.py 파일을 생각해보십시오:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

명령행에서 -h 또는 --help 가 제공되면, ArgumentParser 도움말이 출력됩니다:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

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]

optional arguments:
 --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]

optional arguments:
  +h, ++help  show this help message and exit
```

16.4.3 add_argument() 메서드

ArgumentParser.**add_argument**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

단일 명령행 인자를 구문 분석하는 방법을 정의합니다. 매개 변수마다 아래에서 더 자세히 설명되지만, 요약하면 다음과 같습니다:

- *name or flags* - 옵션 문자열의 이름이나 리스트, 예를 들어 foo 또는 -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 없는 경우 생성되는 값.
- *type* - 명령행 인자가 변환되어야 할 형.
- *choices* - 인자로 허용되는 값의 컨테이너.
- *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.

다음 절에서는 이들 각각의 사용 방법에 관해 설명합니다.

name or flags

add_argument() 메서드는 -f 나 --foo 와 같은 선택 인자가 필요한지, 파일 이름의 리스트와 같은 위치 인자가 필요한지 알아야 합니다. 따라서 *add_argument()* 에 전달되는 첫 번째 인자는 일련의 플래그이거나 간단한 인자 이름이어야 합니다. 예를 들어 선택 인자는 이렇게 만들어질 수 있습니다:

```
>>> 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
```

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' - 인자 값을 저장합니다. 이것이 기본 액션입니다. 예를 들면:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args('--foo 1'.split())
  Namespace(foo='1')
  ```

- 'store_const' - *const* 키워드 인자에 의해 지정된 값을 저장합니다. 'store_const' 액션은 어떤 종류의 플래그를 지정하는 선택 인자와 함께 사용하는 것이 가장 일반적입니다. 예를 들면:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- 'store_true' 와 'store_false' - 각각 True 와 False 값을 저장하는 'store_const' 의 특별한 경우입니다. 또한, 각각 기본값 False 와 True 를 생성합니다. 예를 들면:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_true')
  >>> parser.add_argument('--bar', action='store_false')
  >>> parser.add_argument('--baz', action='store_false')
  >>> parser.parse_args('--foo --bar'.split())
  Namespace(foo=True, bar=False, baz=True)
  ```

- `'append'` - 리스트를 저장하고 각 인자 값을 리스트에 추가합니다. 옵션을 여러 번 지정할 수 있도록 하는 데 유용합니다. 사용 예:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- `'append_const'` - 리스트를 저장하고 *const* 키워드 인자로 지정된 값을 리스트에 추가합니다. (*const* 키워드의 기본값은 `None` 입니다.) `'append_const'` 액션은 여러 개의 인자가 같은 리스트에 상수를 저장해야 할 때 유용합니다. 예를 들면:

```
>>> 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'>])
```

- `'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)
```

  Note, the *default* will be `None` unless explicitly set to *0*.

- `'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
```

Action 서브 클래스나 같은 인터페이스를 구현하는 다른 객체를 전달하여 임의의 액션을 지정할 수도 있습니다. 권장하는 방법은 *Action* 을 확장하여 `__call__` 메서드와 선택적으로 `__init__` 메서드를 재정의하는 것입니다.

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(FooAction, self).__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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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 객체는 일반적으로 하나의 명령행 인자를 하나의 액션과 결합합니다. nargs 키워드 인자는 다른 수의 명령행 인자를 하나의 액션으로 연결합니다. 지원되는 값은 다음과 같습니다:

- 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='?', type=argparse.FileType('r'),
  ...                     default=sys.stdin)
  >>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
  ...                     default=sys.stdout)
  >>> parser.parse_args(['input.txt', 'output.txt'])
  Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
            outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
  >>> parser.parse_args([])
  Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
            outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
  ```

- '*'. 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'])
```

- `'+'`. `'*'` 와 같이, 존재하는 모든 명령행 인자를 리스트로 모읍니다. 또한, 적어도 하나의 명령행 인자가 제공되지 않으면 에러 메시지가 만들어집니다. 예를 들면:

```
>>> 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
```

- `argparse.REMAINDER`. 남은 모든 명령행 인자를 리스트로 수집합니다. 이것은 일반적으로 다른 명령행 유틸리티를 호출하는 명령행 유틸리티에 유용합니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

`nargs` 키워드 인자가 제공되지 않으면, 소비되는 인자의 개수는 *action* 에 의해 결정됩니다. 일반적으로 이는 하나의 명령행 인자가 소비되고 하나의 항목(리스트가 아닙니다)이 생성됨을 의미합니다.

const

add_argument() 의 `const` 인자는 명령행에서 읽지는 않지만 다양한 *ArgumentParser* 액션에 필요한 상숫값을 저장하는 데 사용됩니다. 가장 흔한 두 가지 용도는 다음과 같습니다:

- *add_argument()* 가 `action='store_const'` 또는 `action='append_const'` 로 호출될 때. 이 액션은 *parse_args()* 에 의해 반환된 객체의 어트리뷰트 중 하나에 `const` 값을 추가합니다. 예제는 *action* 설명을 참조하십시오.
- *add_argument()* 가 옵션 문자열(`-f` 또는 `--foo` 와 같은)과 `nargs='?'` 로 호출될 때. 이것은 0 또는 하나의 명령행 인자가 뒤따르는 선택 인자를 만듭니다. 명령행을 파싱할 때 옵션 문자열 뒤에 명령행 인자가 없으면 `const` 값이 대신 가정됩니다. 예제는 *nargs* 설명을 참조하십시오.

`'store_const'` 와 `'append_const'` 액션을 사용할 때는 `const` 키워드 인자를 반드시 주어야 합니다. 다른 액션의 경우, 기본값은 `None` 입니다.

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)
```

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)
```

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 *ArgumentParser* 객체는 명령행 인자를 간단한 문자열로 읽습니다. 그러나 꽤 자주 명령행 문자열은 *float* 또는 *int*와 같은 다른 형으로 해석되어야 합니다. *add_argument()* 의 type 키워드 인자는 필요한 형 검사와 형 변환이 수행되도록 합니다. 일반적인 내장형과 함수는 type 인자의 값으로 직접 사용될 수 있습니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

언제 type 인자가 기본 인자에 적용되는지에 대한 정보는 *default* 키워드 인자 절을 참조하십시오.

다양한 형태의 파일을 사용하기 쉽게 하려고, argparse 모듈은 *open()* 함수의 mode=, bufsize=, encoding=, errors= 인자를 취하는 팩토리 FileType 을 제공합니다. 예를 들어, FileType('w') 은 쓰기 가능한 파일을 만드는 데 사용할 수 있습니다

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

type= 는 단일 문자열 인자를 취하고 변환된 값을 돌려주는 모든 콜러블을 받아들입니다:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

단순히 값의 범위를 검사하는 형 검사기로는 *choices* 키워드 인자가 더 편리할 수 있습니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

자세한 내용은 *choices* 섹션을 참조하십시오.

choices

일부 명령행 인자는 제한된 값 집합에서 선택되어야 합니다. *add_argument()* 에 컨테이너 객체를 *choices* 키워드 인자로 전달하여 처리할 수 있습니다. 명령행을 파싱할 때, 인자의 값을 검사하고, 인자가 받아들일 수 없는 값이 아닌 경우 에러 메시지가 표시됩니다:

```
>>> 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')
```

choices 컨테이너에 포함되는지는 *type* 변환이 수행된 후에 검사하므로, *choices* 컨테이너에 있는 객체의 형은 지정된 *type* 과 일치해야 합니다:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

in 연산자를 지원하는 모든 객체는 *choices* 값으로 전달될 수 있으므로, *dict* 객체, *set* 객체, 사용자 정의 컨테이너 등이 모두 지원됩니다.

required

일반적으로 *argparse* 모듈은 -f 와 --bar 같은 플래그가 명령행에서 생략될 수 있는 선택적 인자를 가리킨다고 가정합니다. 옵션을 필수로 만들기 위해, *add_argument()* 의 required= 키워드 인자에 True 를 지정할 수 있습니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

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

help 값은 인자의 간단한 설명이 들어있는 문자열입니다. 사용자가 도움말을 요청하면 (보통 명령행에서 -h 또는 --help 를 사용합니다), help 설명이 각 인자와 함께 표시됩니다:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

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)

optional arguments:
 -h, --help  show this help message and exit
```

도움말 문자열이 %-포매팅을 지원하기 때문에, 도움말 문자열에 리터럴 `%` 을 표시하려면, `%%` 로 이스케이프 처리해야 합니다.

argparse 는 help 값을 `argparse.SUPPRESS` 로 설정함으로써 특정 옵션에 대한 도움말 엔트리를 감추는 것을 지원합니다:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

metavar

ArgumentParser 가 도움말 메시지를 생성할 때, 기대되는 각 인자를 가리킬 방법이 필요합니다. 기본적으로 ArgumentParser 객체는 *dest* 값을 각 객체의 "이름"으로 사용합니다. 기본적으로 위치 인자 액션의 경우 *dest* 값이 직접 사용되고, 선택 인자 액션의 경우 *dest* 값의 대문자가 사용됩니다. 그래서, `dest='bar'` 인 단일 위치 인자는 `bar` 로 지칭됩니다. 하나의 명령행 인자가 따라와야 하는 단일 선택 인자 `--foo` 는 `FOO` 라고 표시됩니다. 예:

```
>>> 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

optional arguments:
 -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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -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]

optional arguments:
 -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')
```

Action 클래스

Action 클래스는 액션 API를 구현합니다. 액션 API는 명령행에서 인자를 처리하는 콜러블을 반환하는 콜러블 객체입니다. 이 API를 따르는 모든 객체는 `add_argument()` 의 `action` 매개 변수로 전달될 수 있습니다.

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action 객체는 ArgumentParser에서 명령행의 하나 이상의 문자열에서 단일 인자를 파싱하는 데 필요한 정보를 나타내기 위해 사용됩니다. Action 클래스는 두 개의 위치 인자와 *ArgumentParser.add_argument()* 에 전달된 `action` 자신을 제외한 모든 키워드 인자들을 받아들여야 합니다.

Action 인스턴스(또는 `action` 매개 변수로 전달된 콜러블의 반환 값)는 "dest", "option_strings", "default", "type", "required", "help" 등의 어트리뷰트가 정의되어야 합니다. 이러한 어트리뷰트를 정의하는 가장 쉬운 방법은 `Action.__init__` 를 호출하는 것입니다.

Action 인스턴스는 콜러블이어야 하므로, 서브 클래스는 네 개의 매개 변수를 받아들이는 `__call__` 메서드를 재정의해야 합니다:

- `parser` - 이 액션을 포함하는 ArgumentParser 객체.
- `namespace` - *parse_args()* 에 의해 반환될 *Namespace* 객체. 대부분의 액션은 *setattr()* 을 사용하여 이 객체에 어트리뷰트를 추가합니다.
- `values` - 형 변환이 적용된 연관된 명령행 인자. 형 변환은 *add_argument()* 에 전달된 *type* 키워드 인자로 지정됩니다.
- `option_string` - 이 액션을 호출하는 데 사용된 옵션 문자열. `option_string` 인자는 선택적이며, 액션이 위치 인자와 관련되어 있으면 생략됩니다.

`__call__` 메서드는 임의의 액션을 수행 할 수 있습니다만, 일반적으로 `dest` 와 `values` 에 기반하여 `namespace` 에 어트리뷰트를 설정합니다.

16.4.4 parse_args() 메서드

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)

인자 문자열을 객체로 변환하고 namespace의 어트리뷰트로 설정합니다. 값들이 설정된 namespace를 돌려줍니다.

이전의 *add_argument()* 호출이 어떤 객체를 만들고 어떤 식으로 대입할지를 결정합니다. 자세한 내용은 *add_argument()* 설명서를 참조하십시오.

- *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'])
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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)
```

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())
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

때로는 ArgumentParser가 *sys.argv* 의 인자가 아닌 다른 인자를 파싱하는 것이 유용 할 수 있습니다. 문자열 리스트를 *parse_args()* 에 전달하면 됩니다. 대화식 프롬프트에서 테스트할 때 유용합니다:

```
>>> 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 유용할 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'
```

16.4.5 기타 유틸리티

부속 명령

ArgumentParser.**add_subparsers**([*title*][, *description*][, *prog*][, *parser_class*][, *action*][, *option_string*][, *dest*][, *required*][, *help*][, *metavar*])

많은 프로그램은 그 기능을 여러 개의 부속 명령으로 나눕니다. 예를 들어, `svn` 프로그램은 `svn checkout`, `svn update`, `svn commit` 과 같은 부속 명령을 호출 할 수 있습니다. 이런 식으로 기능을 나누는 것은, 프로그램이 다른 명령행 인자를 요구하는 여러 가지 다른 기능을 수행할 때 특히 좋은 생각일 수 있습니다. *ArgumentParser* 는 *add_subparsers()* 메서드로 그러한 부속 명령의 생성을 지원합니다. *add_subparsers()* 메서드는 보통 인자 없이 호출되고 특별한 액션 객체를 돌려줍니다. 이 객체에는 `add_parser()` 라는 하나의 메서드가 있습니다. 이 메서드는 명령 이름과 *ArgumentParser* 생성자 인자를 받고 평소와 같이 수정할 수 있는 *ArgumentParser* 객체를 반환합니다.

매개 변수 설명:

- title - 도움말 출력의 부속 파서 그룹 제목; description이 제공되면 기본적으로 "subcommands", 그렇지 않으면 위치 인자를 위한 제목을 사용합니다
- description - 도움말 출력의 부속 파서 그룹에 대한 설명. 기본값은 `None` 입니다.
- prog - 부속 명령 도움말과 함께 표시될 사용 정보. 기본적으로 프로그램 이름 및 부속 파서 인자 앞에 오는 위치 인자
- parser_class - 부속 파서 인스턴스를 만들 때 사용할 클래스. 기본적으로, 현재 파서의 클래스 (예를 들어 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* - Whether or not a subcommand must be provided, by default `False` (added in 3.7)
- *help* - 도움말 출력의 부속 파서 그룹 도움말, 기본적으로 `None`
- *metavar* - 도움말에서 사용 가능한 부속 명령을 표시하는 문자열. 기본적으로 `None` 이며 {cmd1, cmd2, ..} 형식으로 부속 명령을 표시합니다.

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='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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='XYZ', 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.

마찬가지로, 도움말 메시지가 부속 파서에서 요청되면 해당 파서에 대한 도움말만 인쇄됩니다. 도움말 메시지에는 상위 파서나 형제 파서 메시지는 포함되지 않습니다. (하지만 각 부속 파서 명령에 대한 도움말 메시지는 위와 같이 `add_parser()` 에 `help=` 인자를 주어 지정할 수 있습니다.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -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

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

게다가, add_parser 는 aliases 인자를 추가로 지원하는데, 여러 개의 문자열이 같은 부속 파서를 참조 할 수 있게 해줍니다. 이 예는 svn 와 마찬가지로 checkout 의 약자로 co 라는 별칭을 만듭니다:

```
>>> 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')
```

부속 명령을 처리하는 특히 효과적인 방법의 하나는, *add_subparsers()* 메서드를 *set_defaults()* 호출과 결합하여, 각 부속 파서가 어떤 파이썬 함수를 실행해야 하는지 알 수 있도록 하는 것입니다. 예를 들면:

```
>>> # sub-command 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()
>>>
>>> # 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')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

버전 3.7에서 변경: New *required* keyword argument.

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 객체는 의사 인자 '-' 를 이해하고, 읽기 위한 *FileType* 객체는 sys.stdin 으로, 쓰기 위한 *FileType* 객체는 sys.stdout 으로 자동 변환합니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

버전 3.4에 추가: *encodings* 및 *errors* 키워드 인자

인자 그룹

ArgumentParser.**add_argument_group**(*title=None*, *description=None*)

기본적으로 *ArgumentParser* 는 도움말 메시지를 표시할 때 "positional arguments"(위치 인자)와 "optional arguments"(선택 인자)로 명령행 인자를 그룹화합니다. 이 기본 그룹보다 더 나은 개념적 인자 그룹이 있는 경우, *add_argument_group()* 메서드를 사용하여 적절한 그룹을 만들 수 있습니다.:

```
>>> 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
group:
  bar    bar help
  --foo FOO  foo help
```

add_argument_group() 메서드는 *ArgumentParser* 처럼 *add_argument()* 메서드를 가진 인자 그룹 객체를 반환합니다. 인자가 그룹에 추가될 때, 파서는 일반 인자처럼 취급하지만, 도움말 메시지는 별도의 그룹에 인자를 표시합니다. *add_argument_group()* 메서드는 이 표시를 사용자 정의하는데 사용할 수 있는 *title* 과 *description* 인자를 받아들입니다:

```
>>> 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
```

사용자 정의 그룹에 없는 인자는 일반적인 "positional arguments" 및 "optional arguments" 섹션으로 들어갑니다.

상호 배제

ArgumentParser.**add_mutually_exclusive_group**(*required=False*)

상호 배타적인 그룹을 만듭니다. *argparse* 는 상호 배타적인 그룹에서 오직 하나의 인자만 명령행에 존재하는지 확인합니다:

```
>>> 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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

현재, 상호 배타적인 인자 그룹은 *add_argument_group()* 의 *title* 및 *description* 인자를 지원하지 않습니다.

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*)

때에 따라 스크립트는 명령행 인자 중 일부만 파싱하고 나머지 인자를 다른 스크립트 나 프로그램에 전달할 수 있습니다. 이 경우 *parse_known_args()* 메서드가 유용 할 수 있습니다. *parse_args()* 와 매우 유사하게 작동하는데, 여분의 인자가 있을 때 에러를 발생시키지 않는 점이 다릅니다. 대신, 채워진 이름 공간과 여분의 인자 문자열 리스트를 포함하는 두 항목 튜플을 반환합니다.

```
>>> 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: 접두사 일치 규칙은 parse_known_args() 에 적용됩니다. 파서는 알려진 옵션 중 하나의 접두사일지라도 여분의 인자 목록에 남기지 않고 옵션을 소비할 수 있습니다.

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*)
이 메서드는 지정된 *status* 상태 코드로 프로그램을 종료하고, *message* 가 주어지면 그 전에 인쇄합니다.

ArgumentParser.**error**(*message*)
이 메서드는 *message* 를 포함하는 사용법 메시지를 표준 에러에 인쇄하고, 상태 코드 2로 프로그램을 종료합니다.

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.

이 파서들은 argparse 기능을 모두 지원하지는 않으며 지원되지 않는 기능이 사용되는 경우 예외를 발생시킵니다. 특히 부속 파서, argparse.REMAINDER, 그리고 옵션과 위치를 모두 포함하는 상호 배타적인 그룹은 지원되지 않습니다.

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.

버전 3.7에 추가.

16.4.6 optparse 코드 업그레이드

원래, *argparse* 모듈은 *optparse* 와의 호환성을 유지하려고 시도했습니다. 그러나, *optparse* 는 투명하게 확장하기 어려운데, 특히 새로운 nargs= 지정자와 더 나은 사용법 메시지를 지원하는 데 필요한 변경에서 그렇습니다. *optparse* 에 있는 대부분이 복사-붙여넣기 되었거나 몽키 패치되었을 때, 더 하위 호환성을 유지하려고 노력하는 것이 실용적으로 보이지 않게 되었습니다.

argparse 모듈은 표준 라이브러리 *optparse* 모듈을 다음과 같은 여러 가지 방식으로 개선합니다:

- 위치 인자 처리.
- 부속 명령 지원.
- + 와 / 와 같은 다른 옵션 접두사 허용.
- 0개 이상 및 1개 이상 스타일의 인자 처리.
- 더욱 유익한 사용법 메시지 생성.
- 사용자 정의 type 과 action 을 위한 훨씬 간단한 인터페이스 제공.

optparse 에서 *argparse* 로의 부분적인 업그레이드 경로:

- 모든 *optparse.OptionParser.add_option()* 호출을 *ArgumentParser.add_argument()* 호출로 대체하십시오.
- `(options, args) = parser.parse_args()` 를 `args = parser.parse_args()` 로 대체하고, 위치 인자에 대한 *ArgumentParser.add_argument()* 호출을 추가하십시오. 이전에 `options` 라고 불렀던 것이 이제 *argparse* 문맥에서 `args` 라는 것을 명심하십시오.
- *optparse.OptionParser.disable_interspersed_args()* 를 *parse_args()* 대신 *parse_intermixed_args()* 를 사용하여 대체하십시오.
- 콜백 액션과 `callback_*` 키워드 인자를 `type` 또는 `action` 인자로 대체하십시오.
- `type` 키워드 인자를 위한 문자열 이름을 해당 type 객체(예를 들어, int, float, complex 등)로 대체하십시오.
- `optparse.Values` 를 *Namespace* 로, `optparse.OptionError` 와 `optparse.OptionValueError` 를 `ArgumentError` 로 대체하십시오.
- `%default` 나 `%prog` 와 같은 묵시적인 인자를 포함하는 문자열을, 문자열 포맷에 딕셔너리를 사용하는 표준 파이썬 문법 대체하십시오, 즉 `%(default)s` 와 `%(prog)s`.
- OptionParser 생성자의 `version` 인자를 `parser.add_argument('--version', action='version', version='<the version>')` 호출로 대체하십시오.

16.5 getopt — 명령 줄 옵션용 C 스타일 구문 분석기

소스 코드: Lib/getopt.py

참고: *getopt* 모듈은 API가 C `getopt()` 함수의 사용자에게 익숙하도록 설계된 명령 줄 옵션용 파서입니다. C `getopt()` 함수에 익숙하지 않거나, 더 적은 코드를 작성하고 더 나은 도움말과 에러 메시지를 얻으려는 사용자는 대신 *argparse* 모듈 사용을 고려해야 합니다.

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.

이 모듈은 두 가지 함수와 예외를 제공합니다:

getopt.**getopt**(*args*, *shortopts*, *longopts=[]*)

명령 줄 옵션과 매개 변수 목록을 구문 분석합니다. *args*는 실행 중인 프로그램에 대한 앞머리 참조를 포함하지 않는, 구문 분석할 인자 리스트입니다. 일반적으로, 이는 `sys.argv[1:]`를 의미합니다. *shortopts*는 스크립트가 인식하고자 하는 옵션 문자의 문자열이며, 인자를 요구하는 옵션은 뒤에 콜론(`':'`; 즉, 유닉스 `getopt()` 가 사용하는 것과 같은 형식)이 필요합니다.

> **참고:** GNU `getopt()` 와는 달리, 옵션이 아닌 인자 다음에 오는 모든 인자는 옵션이 아닌 것으로 간주합니다. 이는 비 GNU 유닉스 시스템이 작동하는 방식과 비슷합니다.

지정되면, *longopts*는 지원되어야 하는 긴 옵션의 이름을 가진 문자열 리스트여야 합니다. 선행 `'--'` 문자는 옵션 이름에 포함되지 않아야 합니다. 인자가 필요한 긴 옵션 뒤에는 등호(`'='`)가 와야 합니다. 선택적 인자는 지원되지 않습니다. 긴 옵션만 허용하려면, *shortopts*는 빈 문자열이어야 합니다. 명령 줄에서 긴 옵션은 허용된 옵션 중 하나와 정확히 일치하는 옵션 이름의 접두사를 제공하는 한 인식 할 수 있습니다. 예를 들어, *longopts*가 `['foo', 'frob']` 면 `--fo` 옵션은 `--foo`로 일치하지만, `--f`는 유일하게 일치하지 않으므로 *GetoptError*가 발생합니다.

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.

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_CORRECT가 설정되면, 옵션이 아닌 인자를 만나자마자 옵션 처리가 중지됩니다.

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 opt는 에러 메시지와 관련 옵션을 제공합니다; 예외와 관련된 특정 옵션이 없으면 opt는 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:')
>>> 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']
```

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:
        # print help information and exit:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    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"
    # ...

if __name__ == "__main__":
    main()
```

argparse 모듈을 사용하면 더 적은 코드로, 더욱 유용한 도움말과 에러 메시지를 제공하는 동등한 명령 줄 인터페이스를 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')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

더 보기:

모듈 *argparse* 대안 명령 줄 옵션과 인자 구문 분석 라이브러리.

16.6 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.

이 모듈은 많은 기능과 유연성을 제공합니다. 로깅에 익숙하지 않다면, 감을 잡는 가장 좋은 방법은 자습서를 보는 것입니다 (오른쪽 링크를 참조하세요).

모듈에 의해 정의된 기본 클래스와 그 기능은 다음과 같습니다.

- 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.

16.6.1 Logger 객체

Loggers have the following attributes and methods. Note that Loggers should *NEVER* be instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to *getLogger()* with the same name will always return a reference to the same Logger object.

`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. 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**

propagate
이 어트리뷰트가 참으로 평가되면, 이 로거에 로그 된 이벤트는 이 로거에 첨부된 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.

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.

---

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*)
Indicates if a message of severity *level* would be processed by this logger. This method checks first the module-level level set by `logging.disable(level)` and then the logger's effective level as determined by *getEffectiveLevel()*.

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.

버전 3.2에 추가.

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.)

kwargs 에서 검사되는 세 개의 키워드 인자가 있습니다: *exc_info*, *stack_info* 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 키워드 인자는 *extra* 로, 로깅 이벤트용으로 만들어진 LogRecord의 __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)-15s %(clientip)s %(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 `Formatter` 문서를 보십시오.)

로그 된 메시지에서 이러한 어트리뷰트를 사용하려면 몇 가지 주의를 기울여야 합니다. 위의 예에서, 예를 들어, `Formatter` 에 설정한 포맷 문자열은 LogRecord의 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.

버전 3.2에 추가: *stack_info* 매개 변수가 추가되었습니다.

버전 3.5에서 변경: *exc_info* 매개 변수는 이제 예외 인스턴스를 받아들입니다.

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*)
Logs a message with integer level *level* on this logger. The other arguments are interpreted as for `debug()`.

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*)
Apply this logger's filters to the record and return `True` if the record is to be processed. The filters are

consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

addHandler(*hdlr*)
지정된 처리기 *hdlr* 를 이 로거에 추가합니다.

removeHandler(*hdlr*)
이 로거에서 지정된 처리기 *hdlr* 을 제거합니다.

findCaller(*stack_info=False*)
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.

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.

버전 3.2에 추가.

버전 3.7에서 변경: 이제 로거는 피클 되고 언피클 될 수 있습니다.

16.6.2 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
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

16.6.3 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*)
이 처리기의 *Formatter*를 *fmt* 로 설정합니다.

addFilter(*filter*)
지정된 필터 *filter* 를 이 처리기에 추가합니다.

removeFilter(*filter*)
이 처리기에서 지정된 필터 *filter* 를 제거합니다.

filter(*record*)
Apply this handler's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

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, *shutdown()* 이 호출 될 때 닫히는 처리기의 내부 목록에서 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 시스템

에서 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.

표준으로 포함된 처리기 목록은 *logging.handlers* 를 참조하십시오.

16.6.4 Formatter 객체

Formatter 객체는 다음과 같은 어트리뷰트와 메서드를 가지고 있습니다. 이들은 *LogRecord* 를 (보통) 사람이나 외부 시스템이 해석 할 수 있는 문자열로 변환하는 역할을 합니다. 베이스 *Formatter* 는 포매팅 문자열을 지정할 수 있게 합니다. 아무것도 지정하지 않으면, `'%(message)s'` 이 기본값으로 사용되는데, 단지 로깅 호출에서 제공된 메시지만 포함됩니다. 포맷된 출력에 추가 정보(가령 타임스탬프)를 넣으려면 계속 읽으십시오.

포매터는 *LogRecord* 어트리뷰트에 포함된 정보를 사용하는 포맷 문자열로 초기화될 수 있습니다 - 위에서 언급 한 기본값은 사용자의 메시지와 인자가 *LogRecord* 의 *message* 어트리뷰트로 미리 포맷된다는 사실을 활용합니다. 이 포맷 문자열은 표준 파이썬 %-스타일 매핑 키를 포함합니다. 문자열 포매팅에 대해서 더 많은 정보가 필요하면 *printf* 스타일 문자열 포매팅를 보세요.

LogRecord 에 있는 유용한 매핑 키는 *LogRecord* 어트리뷰트 섹션에 있습니다.

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*)
Formatter 클래스의 새로운 인스턴스를 반환합니다. 인스턴스는 전체 메시지의 포맷 문자열과 메시지의 날짜/시간 부분에 대한 포맷 문자열로 초기화됩니다. *fmt* 가 지정되지 않으면 `'%(message)s'` 가 사용됩니다. *datefmt* 가 지정되지 않으면 *formatTime()* 설명서에 기술된 포맷이 사용됩니다.

style 매개 변수는 '%', '{' 또는 '$' 중 하나일 수 있으며, 포맷 문자열이 데이터와 병합되는 방식을 결정합니다: %-포매팅, *str.format()* 또는 *string.Template* 중 하나를 사용합니다. 로그 메시지에 {- 와 $-포매팅을 사용하는 방법에 대한 자세한 내용은 formatting-styles을 참조하십시오.

버전 3.2에서 변경: *style* 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 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, 쉼표 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.

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.

16.6.5 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*)
지정된 레코드가 로그 됩니까? 아니라면 0을 반환하고, 그렇다면 0이 아닌 값을 반환합니다. 적절하다고 판단되면, 이 메서드는 해당 레코드를 수정할 수 있습니다.

처리기에 첨부된 필터는 이벤트를 처리기가 출력하기 전에 호출되는 반면, 로거에 첨부된 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.

필터는 수준보다 정교한 기준에 따라 레코드를 필터링하는 데 주로 사용되지만, 필터가 첨부되는 처리기나 로거에서 처리되는 모든 레코드를 볼 수 있습니다: 이 특성은, 특정 로거나 처리기가 얼마나 많은 레코드를

처리하는지 센다거나, 처리 중인 LogRecord에 어트리뷰트를 추가, 변경, 삭제하려고 할 때 유용합니다. 당연히, LogRecord를 변경하는 것은 주의를 필요로 하는 일이지만, 로그에 문맥 정보를 주입하는 것을 허용합니다 (filters-contextual를 보세요).

16.6.6 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 msg와 args 로 전달되며, msg % args 를 사용하여 병합되어 레코드의 message 필드를 만듭니다.

매개변수

- **name** – 이 LogRecord가 나타내는 이벤트를 로그 하는데 사용된 로거의 이름. 이 이름은 다른 (조상) 로거에 첨부된 처리기가 출력하더라도 항상 이 값을 갖습니다.
- **level** – 로깅 이벤트의 숫자 수준 (DEBUG, INFO 등). 이 값은 LogRecord의 두 어트리뷰트로 변환됩니다: 숫자 값을 위한 levelno 와 해당 수준 이름을 위한 levelname.
- **pathname** – 로깅 호출이 발생한 소스 파일의 전체 경로명.
- **lineno** – 로깅 호출이 발생한 소스 파일의 행 번호.
- **msg** – 이벤트 설명 메시지. 변수 데이터를 위한 자리 표시자가 있는 포맷 문자열일 수 있습니다.
- **args** – 이벤트 설명을 얻기 위해 *msg* 인자에 병합할 변수 데이터.
- **exc_info** – 현재의 예외 정보를 가지는 예외 튜플. 예외 정보가 없는 경우는 None 입니다.
- **func** – 로깅 호출을 호출한 함수 또는 메서드의 이름.
- **sinfo** – 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.

16.6.7 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:03d}` 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()* 이 반환하는 시간).
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	`%(relativeCreated)d`	logging 모듈이 로드된 시간을 기준으로 LogRecord가 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).

버전 3.1에서 변경: *processName* 이 추가되었습니다.

16.6.8 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*)
하부 *Logger* 인스턴스와 딕셔너리 류 객체로 초기화된 *LoggerAdapter* 의 인스턴스를 반환합니다.

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.

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.

16.6.9 스레드 안전성

로깅 모듈은 클라이언트가 특별한 주의를 기울이지 않아도 스레드 안전하도록 만들어졌습니다. 이렇게 하려고 threading 록을 사용합니다; 모듈의 공유 데이터에 대한 액세스를 직렬화하는 록이 하나 있고, 각 처리기 또한 하부 I/O에 대한 액세스를 직렬화하는 록을 만듭니다.

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.

16.6.10 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 로거인 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 있습니다.

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()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**()
LogRecord 를 생성하는 데 사용되는 콜러블을 반환합니다.

버전 3.2에 추가: 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*)

루트 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.)

kwargs 에서 검사되는 세 개의 키워드 인자가 있습니다: *exc_info* 가 거짓으로 평가되지 않으면, 로깅 메시지에 예외 정보가 추가됩니다. 예외 튜플(*sys.exc_info()* 에 의해 반환되는 형식)이나 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 *extra* 로, 로깅 이벤트용으로 만들어진 LogRecord의 __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)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.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 *Formatter* 문서를 보십시오.)

로그 된 메시지에서 이러한 어트리뷰트를 사용하려면 몇 가지 주의를 기울여야 합니다. 위의 예에서, 예를 들어, *Formatter* 에 설정한 포맷 문자열은 LogRecord의 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.

버전 3.2에 추가: *stack_info* 매개 변수가 추가되었습니다.

logging.**info**(*msg*, **args*, ***kwargs*)

루트 로거에 수준 INFO 메시지를 로그 합니다. 인자는 *debug()*처럼 해석됩니다.

logging.**warning**(*msg*, **args*, ***kwargs*)

루트 로거에 수준 WARNING 메시지를 로그 합니다. 인자는 *debug()*처럼 해석됩니다.

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. 인자는 *debug()* 처럼 해석됩니다.

logging.**critical**(*msg*, **args*, ***kwargs*)
루트 로거에 수준 CRITICAL 메시지를 로그 합니다. 인자는 *debug()* 처럼 해석됩니다.

logging.**exception**(*msg*, **args*, ***kwargs*)
루트 로거에 수준 ERROR 메시지를 로그 합니다. 인자는 *debug()* 처럼 해석됩니다. 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. 다른 인자는 *debug()* 처럼 해석됩니다.

참고: 위의 루트 로거에 위임하는 모듈 수준 편리 함수는 적어도 하나의 처리기를 사용할 수 있도록 *basicConfig()* 를 호출합니다. 이 때문에, 스레드가 시작되기 전에 적어도 하나의 처리기가 루트 로거에 추가되지 않는 한, 2.7.1 및 3.2 이전의 파이썬 버전에서는 스레드에서 이 함수들을 사용하지 않아야 합니다. 이전 버전의 파이썬에서는, *basicConfig()* 의 스레드 안전성 결함으로 인해 (드물긴 하지만) 처리기가 루트 로거에 여러 번 추가될 수 있으며, 같은 이벤트가 여러 번 기록되는 것으로 이어질 수 있습니다.

logging.**disable**(*level=CRITICAL*)
Provides an overriding level *level* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *level* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than CRITICAL (this is not recommended), you won't be able to rely on the default value for the *level* parameter, but will have to explicitly supply a suitable value.

버전 3.7에서 변경: The *level* parameter was defaulted to level CRITICAL. See Issue #28524 for more information about this change.

logging.**addLevelName**(*level*, *levelName*)
Associates level *level* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a *Formatter* formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

참고: 자신만의 수준을 정의할 생각이라면 custom-levels 섹션을 보십시오.

logging.**getLevelName**(*level*)
Returns the textual representation of logging level *level*. If the level is one of the predefined levels CRITICAL, ERROR, WARNING, INFO or DEBUG then you get the corresponding string. If you have associated levels with names using *addLevelName()* then the name you have associated with *level* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string 'Level %s' % level is returned.

참고: 수준은 (로깅 로직에서 비교해야 하므로) 내부적으로 정수입니다. 이 함수는 장수 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.`**`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 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 FileHandler를 만들도록 지정합니다.
filemode	*filename* 이 지정되었으면, 이 모드 로 파일을 엽니다. 기본값은 `'a'` 입니다.
format	처리기에 지정된 포맷 문자열을 사용합니다.
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	StreamHandler의 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.

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.

`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.`**`setLoggerClass`**`(`*klass*`)`

Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()`

such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior. After this call, as at any other time, do not instantiate loggers directly using the subclass: continue to use the *logging.getLogger()* API to get your loggers.

logging.**setLogRecordFactory**(*factory*)
LogRecord 를 만드는데 사용되는 콜러블을 설정합니다.

매개변수 factory – 로그 레코드의 인스턴스를 만드는데 사용되는 팩토리 콜러블.

버전 3.2에 추가: 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.

16.6.11 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.

버전 3.2에 추가.

16.6.12 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.

16.7 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.

16.7.1 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.

버전 3.2에 추가.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*)

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).

매개변수

- **fname** – 파일명, 또는 파일류 객체, 또는 *RawConfigParser*에서 파생된 인스턴스. RawConfigParser-파생 인스턴스가 전달되면, 그대로 사용됩니다. 그렇지 않으면, Configparser의 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.

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.

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 None을 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 반환 값에 대해 `join()`을 호출하기 전에 호출됩니다.

16.7.2 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.

  구성 딕셔너리는 키 `format`과 `datefmt`(기본값은 `None`)으로 검색되며 이들은 *Formatter* 인스턴스를 만드는 데 사용됩니다.

- *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.

  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=3`를 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
  - `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:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    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.baz라는 로거에 두 개의 처리기가 연결되어 있어야 하며, 이 처리기들은 처리기 id h1과 h2``에 의해 기술됩니다. ``h1의 포매터는 id brief로 기술되는 것이고, h2의 포매터는 id precise로 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, brief와 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 지정

하면 일반적인 임포트 메커니즘을 사용하여 실제 콜러블을 얻습니다. 콜러블은 구성 서브 딕셔너리의 **나머지** 항목을 키워드 인자로 호출됩니다. 위의 예제에서, id가 custom인 포매터는 다음과 같은 호출이 반환한다고 가정합니다:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

'()' 키가 유효한 키워드 매개 변수 이름이 아니라서 특수키로 사용되었습니다. 그러므로 호출에 사용되는 키워드 인자의 이름과 충돌하지 않습니다. '()'는 해당 값이 콜러블이라는 표시로도 기능합니다.

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.stderr를 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. prefix가 인식되면 suffix는 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 level에 대한 문자열 값 'DEBUG'은 자동으로 값 logging.DEBUG으로 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 target은 단지 관련 target 처리기의 객체 id이기만 하면 되며, 시스템은 id로부터 처리기를 결정합니다. 그러나 사용자가 alternate 처리기를 갖는 my.package.MyHandler를 정의하면, 구성 시스템은 alternate가 처리기를 참조한다는 것을 알 수 없습니다. 이 문제를 해결하기 위해, 일반 결정 시스템은 사용자가 다음과 같이 지정할 수 있게 합니다:

```
handlers:
  file:
    # configuration of file handler goes here

  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.format에서 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    toaddrs:
      - 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. 인덱스값이 십진수로만 구성되면, 해당 정숫값을 사용하여 액세스가 시도되고, 필요하면 문자열 값으로 다시 시도합니다.

문자열 cfg://handlers.myhandler.mykey.123이 주어지면, config_dict['handlers']['myhandler']['myke으로 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.

16.7.3 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 log01이라고 이름 붙은 로거에 대해, 관련 구성 세부 사항은 [logger_log01] 섹션에 담깁니다. 마찬가지로, [handlers] 섹션에서 hand01이라고 부르는 처리기는 [handler_hand01]이라는 섹션에 구성이 담기고, [formatters] 섹션에서 form01이라고 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

[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, NOTSET는 모든 메시지가 로그 됨을 의미합니다. 수준 값은 logging 패키지의 이름 공간 컨텍스트에서 *eval()* 됩니다.

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
```

level과 handlers 항목은 루트 로거에서처럼 해석됩니다. 단, 루트가 아닌 로거의 수준이 NOTSET로 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. level은 로거에서처럼 해석되며, NOTSET은 '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 *eval()*될 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 *eval()*될 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
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
datefmt=
class=logging.Formatter
```

format 항목은 전체 포맷 문자열이고, datefmt 항목은 strftime() 호환 날짜/시간 포맷 문자열입니다. 비어있으면, 패키지는 날짜 포맷 문자열 '%Y-%m-%d %H:%M:%S'를 지정하는 것과 거의 동등한 것으로 대체합니다. 이 포맷은 밀리 초도 지정하는데, 위의 포맷 문자열을 사용한 결과에 쉼표 구분 기호와 함께 추가됩니다. 이 포맷의 예제 시간은 2003-01-23 00:29:50,411입니다.

class 항목은 선택적입니다. 포매터 클래스의 이름을 나타냅니다(점으로 구분된 모듈과 클래스 이름). 이 옵션은 *Formatter* 서브 클래스를 인스턴스화하는 데 유용합니다. *Formatter*의 서브 클래스는 확장 또는 압축 형식으로 예외 트래이스백을 표시할 수 있습니다.

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.

16.8 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.

16.8.1 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 종결자(terminator)와 함께 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.

반환값 스트림이 변경되면 이전 스트림, 그렇지 않으면 *None*.

버전 3.7에 추가.

버전 3.2에서 변경: StreamHandler 클래스는 이제 포맷된 레코드를 스트림에 쓸 때 종결자로 사용되는 terminator 어트리뷰트(기본 값 '\n')를 갖습니다. 이 개행 문자 종료를 원하지 않는다면, 처리기 인스턴스의 terminator 어트리뷰트를 빈 문자열로 설정할 수 있습니다. 이전 버전에서는, 종결자가 '\n' 로 하드코딩되었습니다.

16.8.2 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*)
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.

버전 3.6에서 변경: 문자열 값뿐만 아니라, *Path* 객체도 *filename* 인자로 허용됩니다.

close()
파일을 닫습니다.

emit(*record*)
레코드를 파일에 출력합니다.

16.8.3 NullHandler

버전 3.1에 추가.

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.

16.8.4 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*)

WatchedFileHandler 클래스의 새 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.

버전 3.6에서 변경: 문자열 값뿐만 아니라, *Path* 객체도 *filename* 인자로 허용됩니다.

reopenIfNeeded()

파일이 변경되었는지 확인합니다. 그렇다면, 기존 스트림을 플러시 한 후 닫고, 파일을 다시 엽니다. 일반적으로 레코드를 파일로 출력하기 전에 수행합니다.

버전 3.6에 추가.

emit(*record*)

레코드를 파일에 출력하지만, 파일이 변경되었을 때 다시 열기 위해 *reopenIfNeeded()* 를 먼저 호출합니다.

16.8.5 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*)

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.

버전 3.3에 추가.

rotator

이 어트리뷰트가 콜러블로 설정되면, *rotate()* 메서드는 이 콜러블에 위임합니다. 콜러블로 전달되는 매개 변수는 *rotate()* 로 전달되는 것입니다.

버전 3.3에 추가.

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.

버전 3.3에 추가.

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'.

버전 3.3에 추가.

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.

16.8.6 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*)
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.

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.2부터 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.

doRollover()
위에서 설명한 대로 롤오버를 수행합니다.

emit(*record*)
앞에서 설명한 대로 롤오버를 처리하면서, 파일에 레코드를 출력합니다.

16.8.7 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*)
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
'midnight'	*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.

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.

doRollover()
위에서 설명한 대로 롤오버를 수행합니다.

emit(*record*)
위에서 설명한 대로 롤오버를 처리하면서, 파일에 레코드를 출력합니다.

16.8.8 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*)
Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket. The details of this operation are equivalent to:

```
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*)
Send a pickled byte-string *packet* to the socket. The format of the sent byte-string is as described in the documentation for *makePickle()*.

This function allows for partial sends, which can happen when the network is busy.

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).

16.8.9 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.

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*)
Send a pickled byte-string to a socket. The format of the sent byte-string is as described in the documentation for *SocketHandler.makePickle()*.

16.8.10 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*)
`(host, port)` 튜플 형태의 *address*로 주어진 주소의 원격 유닉스 기계와 통신하기 위한 *SysLogHandler* 클래스의 새 인스턴스를 돌려줍니다. *address* 를 지정하지 않으면 `('localhost', 514)`가 사용됩니다. 주소는 소켓을 여는 데 사용됩니다. `(host, port)` 튜플을 제공하는 대신, 주소를 문자열로 제공할 수 있습니다, 예를 들어 '/dev/log'. 이 경우, 메시지를 syslog로 보내는데 유닉스 도메인 소켓이 사용됩니다. *facility* 가 지정되지 않으면, `LOG_USER`가 사용됩니다. 열리는 소켓의 유형은 *socktype*

인자에 따라 달라지며, 기본값은 *socket.SOCK_DGRAM*이고, 따라서 UDP 소켓이 열립니다. TCP 소켓을 열려면 (rsyslog와 같은 최신 syslog 데몬을 사용할 때), *socket.SOCK_STREAM* 값을 지정하십시오.

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.

버전 3.2에서 변경: *socktype* 이 추가되었습니다.

close()
원격 호스트로의 소켓을 닫습니다.

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 CRITICAL을 동등한 syslog 이름으로 매핑하고, 다른 모든 수준 이름은 'warning'으로 매핑합니다.

16.8.11 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.

16.8.12 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 3.3에 추가: *timeout* 인자가 추가되었습니다.

emit(*record*)
레코드를 포맷하고 지정된 주소로 보냅니다.

getSubject(*record*)
레코드에 종속적인 제목 줄을 지정하려면, 이 메서드를 재정의하십시오.

16.8.13 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*)
Initializes the handler with a buffer of the specified capacity. Here, *capacity* means the number of logging records buffered.

emit(*record*)
Append the record to the buffer. If `shouldFlush()` returns true, call `flush()` to process the buffer.

flush()
사용자 정의 플러시 동작을 구현하기 위해 재정의할 수 있습니다. 이 버전은 버퍼를 비우기만 합니다.

shouldFlush(*record*)
Return `True` if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)
Returns a new instance of the `MemoryHandler` class. The instance is initialized with a buffer size of *capacity* (number of records buffered). If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using `setTarget()` before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

버전 3.6에서 변경: *flushOnClose* 매개 변수가 추가되었습니다.

close()
`flush()`를 호출하고, 대상(target)을 `None`으로 설정하고, 버퍼를 비웁니다.

flush()
`MemoryHandler`의 경우, 플러시는 버퍼링 된 레코드가 있다면 대상으로 보내는 것을 뜻합니다. 이때 버퍼도 지워집니다. 다른 행동을 원하면 재정의하십시오.

setTarget(*target*)
이 처리기의 대상 처리기를 설정합니다.

shouldFlush(*record*)
버퍼 가득 참이나 레코드가 *flushLevel* 이상을 만드는지 확인합니다.

16.8.14 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.

16.8.15 QueueHandler

버전 3.2에 추가.

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*)

Returns a new instance of the *QueueHandler* class. The instance is initialized with the queue to send messages to. The *queue* can be any queue-like object; it's used as-is by the *enqueue()* method, which needs to know how to send messages to it. The queue is not *required* to have the task tracking API, which means that you can use *SimpleQueue* instances for *queue*.

emit(*record*)

Enqueues the result of preparing the LogRecord. Should an exception occur (e.g. because a bounded queue has filled up), the *handleError()* method is called to handle the error. This can result in the record silently being dropped (if `logging.raiseExceptions` is `False`) or a message printed to `sys.stderr` (if `logging.raiseExceptions` is `True`).

prepare(*record*)

큐에 넣기 위해 레코드를 준비합니다. 이 메서드에 의해 반환된 객체는 큐에 들어갑니다.

기본 구현은 메시지, 인자와 있다면 예외 정보를 병합하도록 레코드를 포맷합니다. 또한, 역 피클할 수 없는 항목들을 레코드에서 직접(in-place) 제거합니다.

레코드를 dict 나 JSON 문자열로 변환하거나, 원본을 그대로 두고 레코드의 수정 된 복사본을 보내길 원한다면 이 메서드를 재정의할 수 있습니다.

enqueue(*record*)

`put_nowait()`를 사용하여 큐에 레코드를 넣습니다; 블로킹 동작이나 시간제한이나, 사용자 정의 큐 구현을 사용하려면 이 메서드를 재정의할 수 있습니다.

16.8.16 QueueListener

버전 3.2에 추가.

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*)

Returns a new instance of the *QueueListener* class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the *dequeue()* method, which needs to know how to get messages from it. The queue is not *required* to have the task tracking API (though it's used if available), which means that you can use *SimpleQueue* instances for *queue*.

If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

버전 3.5에서 변경: The `respect_handler_level` argument was added.

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.

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.

버전 3.3에 추가.

더 보기:

모듈 *logging* logging 모듈에 관한 API 레퍼런스.

모듈 *logging.config* logging 모듈용 구성 API.

16.9 getpass — 이식성 있는 암호 입력

소스 코드: Lib/getpass.py

---

getpass 모듈은 두 함수를 제공합니다:

getpass.**getpass**(*prompt='Password: ', stream=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).

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.

---

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 예외가 발생합니다.

일반적으로, 이 함수는 *os.getlogin()*보다 선호됩니다.

16.10 curses — Terminal handling for character-cell displays

The *curses* module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

참고: Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

참고: Since version 5.4, the ncurses library decides how to interpret non-ASCII data using the `nl_langinfo` function. That means that you have to call *locale.setlocale()* in the application and encode Unicode strings using one of the system's available encodings. This example uses the system's default encoding:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Then use *code* as the encoding for *str.encode()* calls.

더 보기:

Module *curses.ascii* Utilities for working with ASCII characters, regardless of your locale settings.

Module *curses.panel* A panel stack extension that adds depth to curses windows.

Module *curses.textpad* Editable text widget for curses supporting **Emacs**-like bindings.

curses-howto Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The Tools/demo/ directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

16.10.1 Functions

The module *curses* defines the following exception:

exception curses.**error**
Exception raised when a curses library function returns an error.

참고: Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to A_NORMAL.

The module *curses* defines the following functions:

curses.**baudrate**()
Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

curses.**beep**()
Emit a short attention sound.

curses.**can_change_color**()
Return True or False, depending on whether the programmer can change the colors displayed by the terminal.

curses.**cbreak**()
Enter cbreak mode. In cbreak mode (sometimes called "rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first *raw()* then *cbreak()* leaves the terminal in cbreak mode.

curses.**color_content**(*color_number*)
Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between 0 and COLORS. Return a 3-tuple, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

curses.**color_pair**(*color_number*)
Return the attribute value for displaying text in the specified color. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. *pair_number()* is the counterpart to this function.

curses.**curs_set**(*visibility*)
Set the cursor state. *visibility* can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state; otherwise raise an exception. On many terminals, the "visible" mode is an underline cursor and the "very visible" mode is a block cursor.

curses.**def_prog_mode**()
Save the current terminal mode as the "program" mode, the mode when the running program is using curses. (Its counterpart is the "shell" mode, for when the program is not in curses.) Subsequent calls to *reset_prog_mode()* will restore this mode.

curses.**def_shell_mode**()
Save the current terminal mode as the "shell" mode, the mode when the running program is not using curses. (Its counterpart is the "program" mode, when the program is using curses capabilities.) Subsequent calls to *reset_shell_mode()* will restore this mode.

curses.**delay_output**(*ms*)
Insert an *ms* millisecond pause in output.

curses.**doupdate**()
Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The *doupdate()* ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a *noutrefresh()* call after write operations such as *addstr()* have been performed on a window. The normal *refresh()* call is simply noutrefresh() followed by doupdate(); if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing noutrefresh() calls on all windows, followed by a single doupdate().

curses.**echo**()
Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.**endwin**()
De-initialize the library, and return terminal to normal status.

curses.**erasechar**()
Return the user's current erase character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**filter**()
The *filter()* routine, if used, must be called before *initscr()* is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.**flash**()
Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by *beep()*.

curses.**flushinp**()
Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.**getmouse**()
After *getch()* returns KEY_MOUSE to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple (id, x, y, z, bstate). *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where *n* is the button number from 1 to 4: BUTTONn_PRESSED, BUTTONn_RELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT.

curses.**getsyx**()
Return the current coordinates of the virtual screen cursor as a tuple `(y, x)`. If *leaveok* is currently `True`, then return `(-1, -1)`.

curses.**getwin**(*file*)
Read window related data stored in the file by an earlier `putwin()` call. The routine then creates and initializes a new window using that data, returning the new window object.

curses.**has_colors**()
Return `True` if the terminal can display colors; otherwise, return `False`.

curses.**has_ic**()
Return `True` if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.**has_il**()
Return `True` if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.**has_key**(*ch*)
Take a key value *ch*, and return `True` if the current terminal type recognizes a key with that value.

curses.**halfdelay**(*tenths*)
Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, raise an exception if nothing has been typed. The value of *tenths* must be a number between `1` and `255`. Use *nocbreak()* to leave half-delay mode.

curses.**init_color**(*color_number*, *r*, *g*, *b*)
Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between `0` and `COLORS`. Each of *r*, *g*, *b*, must be a value between `0` and `1000`. When *init_color()* is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if *can_change_color()* returns `True`.

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 is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between `0` and `COLORS`. 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**()
Initialize the library. Return a *window* object which represents the whole screen.

> **참고:** If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

curses.**is_term_resized**(*nlines*, *ncols*)
Return `True` if *resize_term()* would modify the window structure, `False` otherwise.

curses.**isendwin**()
Return `True` if *endwin()* has been called (that is, the curses library has been deinitialized).

curses.**keyname**(*k*)
Return the name of the key numbered *k* as a bytes object. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-byte bytes object consisting of a caret

(`b'^'`) followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a bytes object consisting of the prefix `b'M-'` followed by the name of the corresponding ASCII character.

`curses.`**`killchar`**`()`
Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

`curses.`**`longname`**`()`
Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *`initscr()`*.

`curses.`**`meta`**`(`*flag*`)`
If *flag* is `True`, allow 8-bit characters to be input. If *flag* is `False`, allow only 7-bit chars.

`curses.`**`mouseinterval`**`(`*interval*`)`
Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

`curses.`**`mousemask`**`(`*mousemask*`)`
Set the mouse events to be reported, and return a tuple `(availmask, oldmask)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns `0`. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

`curses.`**`napms`**`(`*ms*`)`
Sleep for *ms* milliseconds.

`curses.`**`newpad`**`(`*nlines*, *ncols*`)`
Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The *`refresh()`* and *`noutrefresh()`* methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

`curses.`**`newwin`**`(`*nlines*, *ncols*`)`
`curses.`**`newwin`**`(`*nlines*, *ncols*, *begin_y*, *begin_x*`)`
Return a new *window*, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is *nlines*/*ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

`curses.`**`nl`**`()`
Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

`curses.`**`nocbreak`**`()`
Leave cbreak mode. Return to normal "cooked" mode with line buffering.

`curses.`**`noecho`**`()`
Leave echo mode. Echoing of input characters is turned off.

`curses.`**`nonl`**`()`
Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.**noqiflush**()
When the noqiflush() routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call noqiflush() in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.**noraw**()
Leave raw mode. Return to normal "cooked" mode with line buffering.

curses.**pair_content**(*pair_number*)
Return a tuple (fg, bg) containing the colors for the requested color pair. The value of *pair_number* must be between 1 and COLOR_PAIRS - 1.

curses.**pair_number**(*attr*)
Return the number of the color-pair set by the attribute value *attr*. *color_pair()* is the counterpart to this function.

curses.**putp**(*str*)
Equivalent to tputs(str, 1, putchar); emit the value of a specified terminfo capability for the current terminal. Note that the output of *putp()* always goes to standard output.

curses.**qiflush**([*flag*])
If *flag* is False, the effect is the same as calling *noqiflush()*. If *flag* is True, or no argument is provided, the queues will be flushed when these control characters are read.

curses.**raw**()
Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.**reset_prog_mode**()
Restore the terminal to "program" mode, as previously saved by *def_prog_mode()*.

curses.**reset_shell_mode**()
Restore the terminal to "shell" mode, as previously saved by *def_shell_mode()*.

curses.**resetty**()
Restore the state of the terminal modes to what it was at the last call to *savetty()*.

curses.**resize_term**(*nlines*, *ncols*)
Backend function used by *resizeterm()*, performing most of the work; when resizing the windows, *resize_term()* blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The resize_term() function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.**resizeterm**(*nlines*, *ncols*)
Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.**savetty**()
Save the current state of the terminal modes in a buffer, usable by *resetty()*.

curses.**setsyx**(*y*, *x*)
Set the virtual screen cursor to *y*, *x*. If *y* and *x* are both -1, then *leaveok* is set True.

curses.**setupterm**(*term=None*, *fd=-1*)
Initialize the terminal. *term* is a string giving the terminal name, or None; if omitted or None, the value of the TERM environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied or -1, the file descriptor for sys.stdout will be used.

curses.**start_color**()
Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after *initscr()*.

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the *curses* module, COLORS and COLOR_PAIRS, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.**termattrs**()
Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.**termname**()
Return the value of the environment variable TERM, as a bytes object, truncated to 14 characters.

curses.**tigetflag**(*capname*)
Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value -1 if *capname* is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

curses.**tigetnum**(*capname*)
Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value -2 if *capname* is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

curses.**tigetstr**(*capname*)
Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return None if *capname* is not a terminfo "string capability", or is canceled or absent from the terminal description.

curses.**tparm**(*str*[, ...])
Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `b'\033[6;4H'`, the exact result depending on terminal type.

curses.**typeahead**(*fd*)
Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is -1, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.**unctrl**(*ch*)
Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

curses.**ungetch**(*ch*)
Push *ch* so the next *getch()* will return it.

> **참고:** Only one *ch* can be pushed before `getch()` is called.

curses.**update_lines_cols**()
Update LINES and COLS. Useful for detecting manual screen resize.

버전 3.5에 추가.

curses.**unget_wch**(*ch*)
Push *ch* so the next *get_wch()* will return it.

> **참고:** Only one *ch* can be pushed before `get_wch()` is called.

버전 3.3에 추가.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)
Push a KEY_MOUSE event onto the input queue, associating the given state data with it.

curses.**use_env**(*flag*)
If used, this function should be called before *initscr()* or newterm are called. When *flag* is False, the values of lines and columns specified in the terminfo database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

curses.**use_default_colors**()
Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, init_pair(x, curses.COLOR_RED, -1) initializes, for instance, color pair *x* to a red foreground color on the default background.

curses.**wrapper**(*func*, ...)
Initialize curses and call another callable object, *func*, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to wrapper(). Before calling *func*, wrapper() turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

16.10.2 Window Objects

Window objects, as returned by *initscr()* and *newwin()* above, have the following methods and attributes:

window.**addch**(*ch*[, *attr*])
window.**addch**(*y*, *x*, *ch*[, *attr*])
Paint character *ch* at (y, x) with attributes *attr*, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

참고: Writing outside the window, subwindow, or pad raises a *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

---

window.**addnstr**(*str*, *n*[, *attr*])
window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])
Paint at most *n* characters of the character string *str* at (y, x) with attributes *attr*, overwriting anything previously on the display.

window.**addstr**(*str*[, *attr*])
window.**addstr**(*y*, *x*, *str*[, *attr*])
Paint the character string *str* at (y, x) with attributes *attr*, overwriting anything previously on the display.

참고: Writing outside the window, subwindow, or pad raises *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

---

window.**attroff**(*attr*)
Remove attribute *attr* from the "background" set applied to all writes to the current window.

window.**attron**(*attr*)
Add attribute *attr* from the "background" set applied to all writes to the current window.

window.**attrset**(*attr*)
Set the "background" set of attributes to *attr*. This set is initially 0 (no attributes).

window.**bkgd**(*ch*[, *attr*])
Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

window.**bkgdset**(*ch*[, *attr*])
Set the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])
Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details.

참고: A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

Parameter	Description	Default value
ls	Left side	ACS_VLINE
rs	Right side	ACS_VLINE
ts	Top	ACS_HLINE
bs	Bottom	ACS_HLINE
tl	Upper-left corner	ACS_ULCORNER
tr	Upper-right corner	ACS_URCORNER
bl	Bottom-left corner	ACS_LLCORNER
br	Bottom-right corner	ACS_LRCORNER

window.**box**([*vertch*, *horch*])
Similar to *border()*, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

window.**chgat**(*attr*)
window.**chgat**(*num*, *attr*)
window.**chgat**(*y*, *x*, *attr*)
window.**chgat**(*y*, *x*, *num*, *attr*)
Set the attributes of *num* characters at the current cursor position, or at position (y, x) if supplied. If *num* is not given or is −1, the attribute will be set on all the characters to the end of the line. This function moves cursor to position (y, x) if supplied. The changed line will be touched using the *touchline()* method so that the contents will be redisplayed by the next window refresh.

window.**clear**()
Like *erase()*, but also cause the whole window to be repainted upon next call to *refresh()*.

window.**clearok**(*flag*)
If *flag* is True, the next call to *refresh()* will clear the window completely.

window.**clrtobot**()
Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of

clrtoeol() is performed.

window.**clrtoeol**()
Erase from cursor to the end of the line.

window.**cursyncup**()
Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

window.**delch**([*y*, *x*])
Delete any character at `(y, x)`.

window.**deleteln**()
Delete the line under the cursor. All following lines are moved up by one line.

window.**derwin**(*begin_y*, *begin_x*)
window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
An abbreviation for "derive window", *derwin()* is the same as calling *subwin()*, except that *begin_y* and *begin_x* are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window.**echochar**(*ch*[, *attr*])
Add character *ch* with attribute *attr*, and immediately call *refresh()* on the window.

window.**enclose**(*y*, *x*)
Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning `True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

window.**encoding**
Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with *window.subwin()*. By default, the locale encoding is used (see *locale.getpreferredencoding()*).

버전 3.3에 추가.

window.**erase**()
Clear the window.

window.**getbegyx**()
Return a tuple `(y, x)` of co-ordinates of upper-left corner.

window.**getbkgd**()
Return the given window's current background character/attribute pair.

window.**getch**([*y*, *x*])
Get a character. Note that the integer returned does *not* have to be in ASCII range: function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return `-1` if there is no input, otherwise wait until a key is pressed.

window.**get_wch**([*y*, *x*])
Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.

버전 3.3에 추가.

window.**getkey**([*y*, *x*])
Get a character, returning a string instead of an integer, as *getch()* does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

window.**getmaxyx**()
Return a tuple `(y, x)` of the height and width of the window.

window.**getparyx**()
Return the beginning coordinates of this window relative to its parent window as a tuple `(y, x)`. Return `(-1, -1)` if this window has no parent.

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.

window.**getyx**()
Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

window.**hline**(*ch*, *n*)
window.**hline**(*y*, *x*, *ch*, *n*)
Display a horizontal line starting at `(y, x)` with length *n* consisting of the character *ch*.

window.**idcok**(*flag*)
If *flag* is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.**idlok**(*flag*)
If *flag* is `True`, *curses* will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.**immedok**(*flag*)
If *flag* is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call *refresh()* yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.**inch**([*y*, *x*])
Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.**insch**(*ch*[, *attr*])
window.**insch**(*y*, *x*, *ch*[, *attr*])
Paint character *ch* at `(y, x)` with attributes *attr*, moving the line from position *x* right by one character.

window.**insdelln**(*nlines*)
Insert *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

window.**insertln**()
Insert a blank line under the cursor. All following lines are moved down by one line.

window.**insnstr**(*str*, *n*[, *attr*])
window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])
Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**insstr**(*str*[, *attr*])
window.**insstr**(*y*, *x*, *str*[, *attr*])
Insert a character string (as many characters as will fit on the line) before the character under the cursor. All

characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

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).

window.**is_linetouched**(*line*)
Return True if the specified line was modified since the last call to *refresh()*; otherwise return False. Raise a *curses.error* exception if *line* is not valid for the given window.

window.**is_wintouched**()
Return True if the specified window was modified since the last call to *refresh()*; otherwise return False.

window.**keypad**(*flag*)
If *flag* is True, escape sequences generated by some keys (keypad, function keys) will be interpreted by *curses*. If *flag* is False, escape sequences will be left as is in the input stream.

window.**leaveok**(*flag*)
If *flag* is True, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *flag* is False, cursor will always be at "cursor position" after an update.

window.**move**(*new_y*, *new_x*)
Move cursor to (new_y, new_x).

window.**mvderwin**(*y*, *x*)
Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.**mvwin**(*new_y*, *new_x*)
Move the window so its upper-left corner is at (new_y, new_x).

window.**nodelay**(*flag*)
If *flag* is True, *getch()* will be non-blocking.

window.**notimeout**(*flag*)
If *flag* is True, escape sequences will not be timed out.

If *flag* is False, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.**noutrefresh**()
Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call *doupdate()*.

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overlay()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of `overwrite()` can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

`window.`**`putwin`**(*file*)
Write all data associated with the window into the provided file object. This information can be later retrieved using the `getwin()` function.

`window.`**`redrawln`**(*beg*, *num*)
Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next `refresh()` call.

`window.`**`redrawwin`**()
Touch the entire window, causing it to be completely redrawn on the next `refresh()` call.

`window.`**`refresh`**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])
Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with `newpad()`. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

`window.`**`resize`**(*nlines*, *ncols*)
Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by `bkgdset()`) merged into them.

`window.`**`scroll`**([*lines=1*])
Scroll the screen or scrolling region upward by *lines* lines.

`window.`**`scrollok`**(*flag*)
Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is `False`, the cursor is left on the bottom line. If *flag* is `True`, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok()`.

`window.`**`setscrreg`**(*top*, *bottom*)
Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

`window.`**`standend`**()
Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

`window.`**`standout`**()
Turn on attribute *A_STANDOUT*.

`window.`**`subpad`**(*begin_y*, *begin_x*)
`window.`**`subpad`**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

`window.`**`subwin`**(*begin_y*, *begin_x*)
`window.`**`subwin`**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.**syncdown**()
Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by *refresh()*, so it should almost never be necessary to call it manually.

window.**syncok**(*flag*)
If *flag* is True, then *syncup()* is called automatically whenever there is a change in the window.

window.**syncup**()
Touch all locations in ancestors of the window that have been changed in the window.

window.**timeout**(*delay*)
Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and *getch()* will return −1 if no input is waiting. If *delay* is positive, then *getch()* will block for *delay* milliseconds, and return −1 if there is still no input at the end of that time.

window.**touchline**(*start*, *count*[, *changed*])
Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=True) or unchanged (*changed*=False).

window.**touchwin**()
Pretend the whole window has been changed, for purposes of drawing optimizations.

window.**untouchwin**()
Mark all lines in the window as unchanged since the last call to *refresh()*.

window.**vline**(*ch*, *n*)
window.**vline**(*y*, *x*, *ch*, *n*)
Display a vertical line starting at (y, x) with length *n* consisting of the character *ch*.

16.10.3 Constants

The *curses* module defines the following data members:

curses.**ERR**
Some curses routines that return an integer, such as *getch()*, return *ERR* upon failure.

curses.**OK**
Some curses routines that return an integer, such as *napms()*, return *OK* upon success.

curses.**version**
A bytes object representing the current version of the module. Also available as __version__.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Attribute	Meaning
A_ALTCHARSET	Alternate character set mode
A_BLINK	Blink mode
A_BOLD	Bold mode
A_DIM	Dim mode
A_INVIS	Invisible or blank mode
A_ITALIC	Italic mode
A_NORMAL	Normal attribute
A_PROTECT	Protected mode
A_REVERSE	Reverse background and foreground colors
A_STANDOUT	Standout mode
A_UNDERLINE	Underline mode
A_HORIZONTAL	Horizontal highlight
A_LEFT	Left highlight
A_LOW	Low highlight
A_RIGHT	Right highlight
A_TOP	Top highlight
A_VERTICAL	Vertical highlight
A_CHARTEXT	Bit-mask to extract a character

버전 3.7에 추가: A_ITALIC was added.

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	Meaning
A_ATTRIBUTES	Bit-mask to extract attributes
A_CHARTEXT	Bit-mask to extract a character
A_COLOR	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with KEY_. The exact keycaps available are system dependent.

Key constant	Key
KEY_MIN	Minimum key value
KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow
KEY_UP	Up-arrow
KEY_LEFT	Left-arrow
KEY_RIGHT	Right-arrow
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace (unreliable)
KEY_F0	Function keys. Up to 64 function keys are supported.
KEY_Fn	Value of function key *n*
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

Key constant	Key
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab
KEY_BEG	Beg (beginning)
KEY_CANCEL	Cancel
KEY_CLOSE	Close
KEY_COMMAND	Cmd (command)
KEY_COPY	Copy
KEY_CREATE	Create
KEY_END	End
KEY_EXIT	Exit
KEY_FIND	Find
KEY_HELP	Help
KEY_MARK	Mark
KEY_MESSAGE	Message
KEY_MOVE	Move
KEY_NEXT	Next
KEY_OPEN	Open
KEY_OPTIONS	Options
KEY_PREVIOUS	Prev (previous)
KEY_REDO	Redo
KEY_REFERENCE	Ref (reference)
KEY_REFRESH	Refresh
KEY_REPLACE	Replace
KEY_RESTART	Restart
KEY_RESUME	Resume
KEY_SAVE	Save
KEY_SBEG	Shifted Beg (beginning)
KEY_SCANCEL	Shifted Cancel
KEY_SCOMMAND	Shifted Command
KEY_SCOPY	Shifted Copy
KEY_SCREATE	Shifted Create
KEY_SDC	Shifted Delete char

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

Key constant	Key
KEY_SDL	Shifted Delete line
KEY_SELECT	Select
KEY_SEND	Shifted End
KEY_SEOL	Shifted Clear line
KEY_SEXIT	Shifted Exit
KEY_SFIND	Shifted Find
KEY_SHELP	Shifted Help
KEY_SHOME	Shifted Home
KEY_SIC	Shifted Input
KEY_SLEFT	Shifted Left arrow
KEY_SMESSAGE	Shifted Message
KEY_SMOVE	Shifted Move
KEY_SNEXT	Shifted Next
KEY_SOPTIONS	Shifted Options
KEY_SPREVIOUS	Shifted Prev
KEY_SPRINT	Shifted Print
KEY_SREDO	Shifted Redo
KEY_SREPLACE	Shifted Replace
KEY_SRIGHT	Shifted Right arrow
KEY_SRSUME	Shifted Resume
KEY_SSAVE	Shifted Save
KEY_SSUSPEND	Shifted Suspend
KEY_SUNDO	Shifted Undo
KEY_SUSPEND	Suspend
KEY_UNDO	Undo
KEY_MOUSE	Mouse event has occurred
KEY_RESIZE	Terminal resize event
KEY_MAX	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

Keycap	Constant
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

참고: These are available only after *initscr()* has been called.

ACS code	Meaning
ACS_BBSS	alternate name for upper right corner
ACS_BLOCK	solid square block
ACS_BOARD	board of squares
ACS_BSBS	alternate name for horizontal line
ACS_BSSB	alternate name for upper left corner
ACS_BSSS	alternate name for top tee
ACS_BTEE	bottom tee
ACS_BULLET	bullet
ACS_CKBOARD	checker board (stipple)
ACS_DARROW	arrow pointing down
ACS_DEGREE	degree symbol
ACS_DIAMOND	diamond
ACS_GEQUAL	greater-than-or-equal-to
ACS_HLINE	horizontal line
ACS_LANTERN	lantern symbol
ACS_LARROW	left arrow
ACS_LEQUAL	less-than-or-equal-to
ACS_LLCORNER	lower left-hand corner
ACS_LRCORNER	lower right-hand corner
ACS_LTEE	left tee
ACS_NEQUAL	not-equal sign
ACS_PI	letter pi
ACS_PLMINUS	plus-or-minus sign
ACS_PLUS	big plus sign
ACS_RARROW	right arrow
ACS_RTEE	right tee
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_SBBS	alternate name for lower right corner
ACS_SBSB	alternate name for vertical line
ACS_SBSS	alternate name for right tee
ACS_SSBB	alternate name for lower left corner
ACS_SSBS	alternate name for bottom tee
ACS_SSSB	alternate name for left tee
ACS_SSSS	alternate name for crossover or big plus
ACS_STERLING	pound sterling
ACS_TTEE	top tee
ACS_UARROW	up arrow
ACS_ULCORNER	upper left corner
ACS_URCORNER	upper right corner
ACS_VLINE	vertical line

The following table lists the predefined colors:

Constant	Color
COLOR_BLACK	Black
COLOR_BLUE	Blue
COLOR_CYAN	Cyan (light greenish blue)
COLOR_GREEN	Green
COLOR_MAGENTA	Magenta (purplish red)
COLOR_RED	Red
COLOR_WHITE	White
COLOR_YELLOW	Yellow

16.11 curses.textpad — Text input widget for curses programs

The *curses.textpad* module provides a *Textbox* class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module *curses.textpad* defines the following function:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)
: Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

16.11.1 Textbox objects

You can instantiate a *Textbox* object as follows:

class curses.textpad.**Textbox**(*win*)
: Return a textbox widget object. The *win* argument should be a curses *window* object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates `(0, 0)`. The instance's *stripspaces* flag is initially on.

  Textbox objects have the following methods:

  edit([*validator*])
  : This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* attribute.

  do_command(*ch*)
  : Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	Action
Control-A	Go to left edge of window.
Control-B	Cursor left, wrapping to previous line if appropriate.
Control-D	Delete character under cursor.
Control-E	Go to right edge (stripspaces off) or end of line (stripspaces on).
Control-F	Cursor right, wrapping to next line when appropriate.
Control-G	Terminate, returning the window contents.
Control-H	Delete character backward.
Control-J	Terminate if the window is 1 line, otherwise insert newline.
Control-K	If line is blank, delete it, otherwise clear to end of line.
Control-L	Refresh screen.
Control-N	Cursor down; move down one line.
Control-O	Insert a blank line at cursor location.
Control-P	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

Constant	Keystroke
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()
Return the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* member.

stripspaces
This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

16.12 curses.ascii — Utilities for ASCII characters

The *curses.ascii* module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

Name	Meaning
NUL	
SOH	Start of heading, console interrupt
STX	Start of text
ETX	End of text
EOT	End of transmission

다음 페이지에 계속

표 3 – 이전 페이지에서 계속

Name	Meaning
ENQ	Enquiry, goes with ACK flow control
ACK	Acknowledgement
BEL	Bell
BS	Backspace
TAB	Tab
HT	Alias for TAB: “Horizontal tab”
LF	Line feed
NL	Alias for LF: “New line”
VT	Vertical tab
FF	Form feed
CR	Carriage return
SO	Shift-out, begin alternate character set
SI	Shift-in, resume default character set
DLE	Data-link escape
DC1	XON, for flow control
DC2	Device control 2, block-mode flow control
DC3	XOFF, for flow control
DC4	Device control 4
NAK	Negative acknowledgement
SYN	Synchronous idle
ETB	End transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator, block-mode terminator
US	Unit separator
SP	Space
DEL	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

curses.ascii.**isalnum**(*c*)
Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c) or isdigit(c)`.

curses.ascii.**isalpha**(*c*)
Checks for an ASCII alphabetic character; it is equivalent to `isupper(c) or islower(c)`.

curses.ascii.**isascii**(*c*)
Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.**isblank**(*c*)
Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.**iscntrl**(*c*)
Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.**isdigit**(*c*)
Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to `c in string.digits`.

curses.ascii.**isgraph**(*c*)
Checks for ASCII any printable character except space.

curses.ascii.**islower**(*c*)
Checks for an ASCII lower-case character.

curses.ascii.**isprint**(*c*)
Checks for any ASCII printable character including space.

curses.ascii.**ispunct**(*c*)
Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.**isspace**(*c*)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.**isupper**(*c*)
Checks for an ASCII uppercase letter.

curses.ascii.**isxdigit**(*c*)
Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

curses.ascii.**isctrl**(*c*)
Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.**ismeta**(*c*)
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or single-character strings; when the argument is a string, it is first converted using the built-in function *ord()*.

Note that all these functions check ordinal bit values derived from the character of the string you pass in; they do not actually know anything about the host machine's character encoding.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.**ascii**(*c*)
Return the ASCII value corresponding to the low 7 bits of *c*.

curses.ascii.**ctrl**(*c*)
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.**alt**(*c*)
Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.**unctrl**(*c*)
Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

curses.ascii.**controlnames**
A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic `SP` for the space character.

16.13 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.

16.13.1 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.

16.13.2 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.

16.14 platform — 하부 플랫폼의 식별 데이터에 대한 액세스

소스 코드: Lib/platform.py

참고: 각 플랫폼은 알파벳순으로 나열되고, 리눅스는 유닉스 절에 포함됩니다.

16.14.1 크로스 플랫폼

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: Mac OS X(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 'i386'. 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=0, terse=0*)
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.

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.version과 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**()
Returns the system/OS name, such as 'Linux', 'Darwin', 'Java', 'Windows'. An empty string is returned if the value cannot be determined.

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.

platform.**uname**()
꽤 이식성 있는 uname 인터페이스. *system*, *node*, *release*, *version*, *machine*, *processor*의 6개의 어트리뷰트를 포함한 *namedtuple()*를 반환합니다.

이것이 *os.uname()* 결과에 없는 여섯 번째 어트리뷰트(*processor*)를 추가한다는 것에 유의하십시오. 또한, 어트리뷰트 이름은 처음 두 어트리뷰트에서 다릅니다; *os.uname()*의 이름은 sysname과 nodename입니다.

결정할 수 없는 항목은 ''로 설정됩니다.

버전 3.3에서 변경: 결과가 튜플에서 네임드 튜플로 변경되었습니다.

16.14.2 자바 플랫폼

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).

16.14.3 윈도우 플랫폼

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.

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.

참고: 이 함수는 Mark Hammond의 `win32all` 패키지가 설치되었을 때 가장 잘 작동하지만, 파이썬 2.3 이상에서도 작동합니다(파이썬 2.6에서 이 지원이 추가되었습니다). 당연히 Win32 호환 플랫폼에서만 실행됩니다.

Win95/98 specific

platform.**popen**(*cmd*, *mode='r'*, *bufsize=-1*)
Portable *popen()* interface. Find a working popen implementation preferring `win32pipe.popen()`. On Windows NT, `win32pipe.popen()` should work; on Windows 9x it hangs due to bugs in the MS C library.

버전 3.3부터 폐지: This function is obsolete. Use the *subprocess* module. Check especially the *Replacing Older Functions with the subprocess Module* section.

16.14.4 Mac OS 플랫폼

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)
Mac 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.

16.14.5 유닉스 플랫폼

platform.**dist**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*)
This is another name for *linux_distribution()*.

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

platform.**linux_distribution**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*, *full_distribution_name=1*)
Tries to determine the name of the Linux OS distribution name.

`supported_dists` may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If `full_distribution_name` is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from `supported_dists` is used.

Returns a tuple `(distname,version,id)` which defaults to the args given as parameters. `id` is the item in parentheses after the version number. It is usually the version codename.

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

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.

16.15 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, 꽤 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 – 연산이 허용되지 않습니다

errno.**ENOENT**
No such file or directory – 그런 파일이나 디렉터리가 없습니다

errno.**ESRCH**
No such process – 그런 프로세스가 없습니다

errno.**EINTR**
Interrupted system call – 중단된 시스템 호출

더 보기:

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

errno.**EAGAIN**
Try again – 다시 시도하십시오

errno.**ENOMEM**
Out of memory – 메모리 부족

errno.**EACCES**
Permission denied – 사용 권한이 거부되었습니다

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

errno.**EXDEV**
Cross-device link – 장치 간 링크

errno.**ENODEV**
No such device – 그런 장치가 없습니다

errno.**ENOTDIR**
Not a directory – 디렉터리가 아닙니다

errno.**EISDIR**
Is a directory – 디렉터리입니다

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

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

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.**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

errno.**ECONNRESET**
Connection reset by peer – 피어에 의한 연결 재설정

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.

errno.**ETOOMANYREFS**
Too many references: cannot splice – 참조가 너무 많습니다: 연결할 수 없습니다

errno.**ETIMEDOUT**
Connection timed out – 연결 시간이 초과하였습니다

errno.**ECONNREFUSED**
Connection refused – 연결이 거부되었습니다

errno.**EHOSTDOWN**
Host is down – 호스트가 다운되었습니다

errno.**EHOSTUNREACH**
No route to host – 호스트로 가는 길이 없습니다

errno.**EALREADY**
Operation already in progress – 이미 진행 중인 연산

errno.**EINPROGRESS**
Operation now in progress – 이제 연산이 진행 중입니다

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

16.16 ctypes — 파이썬용 외부 함수 라이브러리

*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.

16.16.1 ctypes 자습서

참고: 이 자습서의 코드 예제는 *doctest*를 사용하여 실제로 작동하는지 확인합니다. 일부 코드 예제는 리눅스, 윈도우 또는 맥 OS X에서 다르게 동작하므로, 주석에 doctest 지시문이 포함되어 있습니다.

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.

이 객체의 어트리뷰트를 액세스하여 라이브러리를 로드합니다. *cdll*은 표준 `cdecl` 호출 규칙을 사용하는 함수를 내보내는 라이브러리를 로드하는 반면, *windll* 라이브러리는 `stdcall` 호출 규칙을 사용하여 함수를 호출합니다. *oledll* 또한 `stdcall` 호출 규칙을 사용하고, 함수가 윈도우 `HRESULT` 에러 코드를 반환한다고 가정합니다. 에러 코드는 함수 호출이 실패할 때 *OSError* 예외를 자동으로 발생시키는 데 사용됩니다.

버전 3.3에서 변경: 윈도우 에러는 *WindowsError*를 일으켜왔습니다. 이제는 *OSError*의 별칭입니다.

다음은 윈도우 용 예제입니다. `msvcrt`는 대부분 표준 C 함수가 포함된 MS 표준 C 라이브러리며, cdecl 호출 규칙을 사용합니다:

```
>>> 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.

리눅스에서, 라이브러리를 로드하기 위해서는 확장자를 포함하는 파일명을 지정해야 하므로, 어트리뷰트 액세스를 사용하여 라이브러리를 로드 할 수 없습니다. dll 로더의 `LoadLibrary()` 메서드를 사용하거나 CDLL의 생성자를 호출하여 인스턴스를 만들어 라이브러리를 로드해야 합니다:

```
>>> 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:

```
>>> from ctypes import *
>>> 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
>>>
```

kernel32 와 user32와 같은 win32 시스템 dll은 종종 ANSI뿐만 아니라 UNICODE 버전의 함수를 내보냅니다. UNICODE 버전은 이름에 W가 추가된 상태로 내보내지고, ANSI 버전은 이름에 A가 추가되어 내보내 집니다. 지정된 모듈 이름의 모듈 핸들을 반환하는 win32 GetModuleHandle 함수는, 다음과 같은 C 프로토타입을 가지며, UNICODE가 정의되어 있는지에 따라 그중 하나를 GetModuleHandle로 노출하기 위해 매크로가 사용됩니다:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

*windll*는 마술적으로 이 중 하나를 선택하려고 하지 않으므로, GetModuleHandleA 나 GetModuleHandleW를 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

다른 파이썬 콜러블처럼 이 함수를 호출할 수 있습니다. 이 예제에서는 시스템 시간을 유닉스 에포크부터의 초로 반환하는 time() 함수와 win32 모듈 핸들을 반환하는 GetModuleHandleA() 함수를 사용합니다.

This example calls both functions with a NULL pointer (None should be used as the NULL pointer):

```
>>> print(libc.time(None))
1150640792
>>> 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)
>>>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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, 정수, 바이트열 객체 및 (유니코드) 문자열은 이러한 함수 호출에서 매개 변수로 직접 사용할 수 있는 유일한 파이썬 자체의 객체입니다. None는 C NULL 포인터로 전달되고, 바이트열 객체와 문자열은 데이터가 저장된 메모리 블록에 대한 포인터로 전달됩니다 (char * 이나 wchar_t *). 파이썬 정수는 플랫폼의 기본 C int 형으로 전달되며, 그 값은 C 형에 맞게 마스크 됩니다.

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_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 나 long long	int
c_ulonglong	unsigned __int64 나 unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t 나 Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL 종료됨)	바이트열 객체나 None
c_wchar_p	wchar_t * (NUL 종료됨)	문자열이나 None
c_void_p	void *	int 나 None

(1) 생성자는 논릿값을 가진 모든 객체를 받아들입니다.

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
>>>
```

c_char_p, *c_wchar_p* 및 *c_void_p* 포인터형의 인스턴스에 새 값을 대입하면 포인터가 가리키는 메모리 위치가 변경됩니다, 메모리 블록의 내용이 아닙니다 (당연히 아닙니다, 파이썬 바이트열 객체는 불변입니다):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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'
>>> 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'
>>>
```

create_string_buffer() 함수는 이전 ctypes 배포에 있는 c_string() 함수뿐만 아니라 c_buffer() 함수(아직 별칭으로 사용할 수 있습니다)를 대체합니다. C 형 wchar_t의 유니코드 문자를 포함하는 가변 메모리 블록을 생성하려면 *create_unicode_buffer()* 함수를 사용하십시오.

함수 호출하기, 계속

printf는 *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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.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
>>>
```

사용자 정의 데이터형을 사용하여 함수 호출하기

또한 *ctypes* 인자 변환을 사용자 정의하여 사용자 고유 클래스의 인스턴스를 함수 인자로 사용할 수 있습니다. *ctypes*는 _as_parameter_ 어트리뷰트를 찾고, 이를 함수 인자로 사용합니다. 물론 정수, 문자열 또는 바이트열 중 하나여야 합니다:

```
>>> 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
>>>
```

_as_parameter_ 인스턴스 변수에 인스턴스의 데이터를 저장하지 않으려면, *property*를 정의하여 요청 시 어트리뷰트를 사용할 수 있게 할 수 있습니다.

필수 인자 형 (함수 프로토타입) 지정하기

argtypes 어트리뷰트를 설정하여 DLL에서 내보낸 함수의 필수 인자 형을 지정할 수 있습니다.

argtypes는 C 데이터형의 시퀀스 여야 합니다 (printf 함수는 포맷 문자열에 따라 개수와 형이 다른 매개변수를 받아들이기 때문에, 여기서는 좋은 예가 아닐 수 있습니다. 반면에 이 기능을 실험하기에 매우 편리하기도 합니다):

```
>>> 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>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
X 2 3.000000
13
>>>
```

함수 호출에 전달하는 여러분 자신의 클래스를 정의했으면, argtypes 시퀀스에서 해당 클래스를 사용할 수 있도록, from_param() 클래스 메서드를 구현해야 합니다. from_param() 클래스 메서드는 함수 호출에 전달된 파이썬 객체를 받습니다. 형 검사나 이 객체가 수용 가능한지 확인하는 데 필요한 모든 작업을 수행한 다음, 객체 자체나 _as_parameter_ 어트리뷰트나 무엇이건 이 경우에 C 함수 인자로 전달되길 원하는 것을 반환해야 합니다. 다시 말하지만, 결과는 정수, 문자열, 바이트열, *ctypes* 인스턴스 또는 _as_parameter_ 어트리뷰트가 있는 객체여야 합니다.

반환형

기본적으로 함수는 C int 형을 반환한다고 가정합니다. 다른 반환형은 함수 객체의 restype 어트리뷰트를 설정하여 지정할 수 있습니다.

다음은 더 고급 예제입니다. strchr 함수를 사용하는데, 문자열 포인터와 char을 기대하고, 문자열에 대한 포인터를 반환합니다:

```
>>> 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
>>>
```

위의 ord("x") 호출을 피하려면, argtypes 어트리뷰트를 설정할 수 있으며, 두 번째 인자는 한 글자 파이썬 바이트열 객체에서 C char로 변환됩니다:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

외부 함수가 정수를 반환하면, 콜러블 파이썬 객체(예를 들어, 함수나 클래스)를 restype 어트리뷰트로 사용할 수도 있습니다. 콜러블은 C 함수가 돌려주는 정수로 호출되며, 이 호출의 결과는 함수 호출의 결과로 사용됩니다. 이것은 에러 반환 값을 검사하고 자동으로 예외를 발생시키는 데 유용합니다:

```
>>> 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.
>>>
```

WinError는 윈도우 FormatMessage() api를 호출하여 에러 코드의 문자열 표현을 가져오고, 예외를 반환하는 함수입니다. WinError는 선택적 에러 코드 매개 변수를 취합니다, 제공하지 않으면 *GetLastError()*를 호출하여 에러 코드를 가져옵니다.

훨씬 더 강력한 에러 검사 메커니즘을 errcheck 어트리뷰트를 통해 사용할 수 있음에 유의하십시오; 자세한 내용은 레퍼런스 설명서를 참조하십시오.

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

구조체와 공용체는 *ctypes* 모듈에 정의된 *Structure*와 *Union* 베이스 클래스에서 파생되어야 합니다. 각 서브 클래스는 _fields_ 어트리뷰트를 정의해야 합니다. _fields_는 필드 이름 과 필드형을 포함하는 2-튜플의 리스트여야 합니다.

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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> 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))
```

필드 디스크립터는 클래스에서 조회할 수 있습니다. 유용한 정보를 제공할 수 있으므로 디버깅에 유용합니다:

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, 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.

구조체/공용체 정렬과 바이트 순서

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior by specifying a _pack_ class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what `#pragma pack(n)` also does in MSVC.

*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

비트 필드를 포함하는 구조체와 공용체를 만드는 것이 가능합니다. 비트 필드는 정수 필드에만 가능하며, 비트 폭은 _fields_ 튜플의 세 번째 항목으로 지정됩니다:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

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
>>> 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 NULL인지 확인합니다(하지만 NULL이 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
>>>
```

형 변환

일반적으로, ctypes는 엄격한 형 검사를 수행합니다. 이는 함수의 argtypes 목록에 POINTER(c_int)가 있거나, 구조체 멤버 필드의 형이 그런 형이라면, 정확히 같은 형의 인스턴스만 허용됨을 뜻합니다. 이 규칙에는 ctypes가 다른 객체를 허용하는 몇 가지 예외가 있습니다. 예를 들어, 포인터형 대신 호환 가능한 배열 인스턴스를 전달할 수 있습니다. 따라서 POINTER(c_int)의 경우, ctype은 c_int 배열을 허용합니다:

```
>>> 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
>>>
```

또한, 함수 인자가 argtypes에 포인터형(가령 POINTER(c_int))으로 명시적으로 선언되면, 대상형(이 경우 c_int)의 객체를 함수에 전달할 수 있습니다. 이때 ctypes는 필요한 *byref()* 변환을 자동으로 적용합니다.

POINTER 형 필드를 NULL로 설정하려면, None을 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
>>>
```

새 `class cell`은 클래스 문 자체에서 사용할 수 없기 때문입니다. *ctypes*에서는, `cell` 클래스를 정의한 다음, class 문 뒤에서 `_fields_` 어트리뷰트를 설정할 수 있습니다:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Let's try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "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.

콜백 함수의 도움을 받아 항목을 정렬하는 데 사용되는 표준 C 라이브러리의 qsort() 함수를 사용하는 예제를 제시합니다. qsort()는 정수 배열을 정렬하는 데 사용될 것입니다:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

qsort()는 정렬 할 데이터에 대한 포인터, 데이터 배열의 항목 수, 항목 하나의 크기 및 비교 함수에 대한 포인터인 콜백으로 호출해야 합니다. 콜백은 항목에 대한 두 개의 포인터로 호출되며, 첫 번째 항목이 두 번째 항목보다 작으면 음의 정수를, 같으면 0을, 그렇지 않으면 양수 정수를 반환해야 합니다.

따라서 콜백 함수는 정수에 대한 포인터들을 받고 정수를 반환해야 합니다. 먼저 콜백 함수를 위한 형을 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]
...
>>>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(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
>>>
```

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

일부 공유 라이브러리는 함수를 내보낼 뿐만 아니라 변수도 내보냅니다. 파이썬 라이브러리 자체에 있는 예는 `Py_OptimizeFlag` 인데, 시작 시 주어진 `-O`나 `-OO` 플래그에 따라, 0, 1 또는 2로 설정된 정수입니다.

*ctypes*는 형의 `in_dll()` 클래스 메서드를 사용하여 이와 같은 값을 액세스할 수 있습니다. *pythonapi*는 파이썬 C API에 대한 액세스를 제공하는 미리 정의된 심볼입니다:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

인터프리터가 `-O`로 시작되면, 예는 `c_long(1)`를, `-OO`가 지정되면 `c_long(2)`를 인쇄합니다.

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 struct _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)]
...
>>>
```

struct _frozen 데이터형을 정의했으므로, 테이블에 대한 포인터를 얻을 수 있습니다:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Since table is a pointer to the array of struct_frozen records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with import __hello__ for example.

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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     _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
>>> # 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.

Another example that may behave differently from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

참고: Objects instantiated from *c_char_p* can only have their value set to bytes or integers.

`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 내용을 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
>>> 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.

16.16.2 ctypes 레퍼런스

공유 라이브러리 찾기

컴파일 언어로 프로그래밍할 때, 공유 라이브러리는 프로그램을 컴파일/링크할 때와 프로그램을 실행할 때 액세스 됩니다.

`find_library()` 함수의 목적은 컴파일러나 실행 시간 로더가 하는 것과 비슷한 방식으로 라이브러리를 찾는 것입니다 (여러 버전의 공유 라이브러리가 있는 플랫폼에서는 가장 최근의 것을 로드해야 합니다). 반면에 ctypes 라이브러리 로더는 프로그램이 실행될 때처럼 동작하고, 실행 시간 로더를 직접 호출합니다.

`ctypes.util` 모듈은 로드할 라이브러리를 판별하는 데 도움이 되는 함수를 제공합니다.

`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.

리눅스에서, `find_library()`는 외부 프로그램(`/sbin/ldconfig`, `gcc`, `objdump` 및 `ld`)을 실행하여 라이브러리 파일을 찾으려고 합니다. 라이브러리 파일의 파일명을 반환합니다.

버전 3.6에서 변경: 리눅스에서, 다른 수단으로 라이브러리를 찾을 수 없으면, 라이브러리 검색 시 환경 변수 LD_LIBRARY_PATH의 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'
>>>
```

OS X에서, find_library()는 라이브러리를 찾기 위해 미리 정의된 몇 가지 명명 체계와 경로를 시도하고 성공하면 전체 경로명을 반환합니다:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/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'
>>>
```

윈도우에서, find_library()는 시스템 검색 경로를 따라 검색하고 전체 경로명을 반환하지만, 미리 정의된 명명 체계가 없으므로 find_library("c")와 같은 호출은 실패하고 None을 반환합니다.

공유 라이브러리를 *ctypes*로 래핑하려면, 실행 시간에 라이브러리를 찾기 위해 find_library()를 사용하기보다, 개발 시점에 공유 라이브러리 이름을 확인하고 래퍼 모듈에 하드 코딩 하는 것이 더 좋을 수 있습니다.

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*)

이 클래스의 인스턴스는 로드된 공유 라이브러리를 나타냅니다. 이 라이브러리의 함수는 표준 C 호출 규칙을 사용하며, int를 반환한다고 가정합니다.

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

윈도우 전용: 이 클래스의 인스턴스는 로드된 공유 라이브러리를 나타내며, 이 라이브러리의 함수는 stdcall 호출 규칙을 사용하고, 윈도우 특정 *HRESULT* 코드를 반환한다고 가정합니다. *HRESULT* 값에는 함수 호출이 실패했는지 또는 성공했는지와 추가 에러 코드를 지정하는 정보가 들어 있습니다. 반환 값이 실패를 알리면, *OSError*가 자동으로 발생합니다.

버전 3.3에서 변경: *WindowsError*를 발생시켰었습니다.

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

윈도우 전용: 이 클래스의 인스턴스는 로드된 공유 라이브러리를 나타내며, 이 라이브러리의 함수는 stdcall 호출 규칙을 사용하고, 기본적으로 int를 반환한다고 가정합니다.

윈도우 CE에서는 오직 표준 호출 규칙만 사용됩니다. 편의상 *WinDLL*와 *OleDLL*은, 이 플랫폼에서 표준 호출 규칙을 사용합니다.

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 릴리스 되지 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.

이러한 모든 클래스는 공유 라이브러리의 경로명인 적어도 하나의 인자를 사용하여 인스턴스를 만들 수 있습니다. 이미 로드된 공유 라이브러리에 대한 기존 핸들이 있으면, 이를 `handle` 이라고 이름 붙은 매개 변수로 전달할 수 있습니다. 그렇지 않으면 하부 플랫폼의 `dlopen` 이나 `LoadLibrary` 함수를 사용하여 라이브러리를 프로세스에 로드하고 이에 대한 핸들을 확보합니다.

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.

use_errno 매개 변수를 참으로 설정하면 시스템 *errno* 에러 번호에 안전하게 액세스할 수 있는 ctypes 메커니즘을 활성화합니다. *ctypes*는 시스템 *errno* 변수의 스레드 로컬 사본을 유지합니다; `use_errno=True`로 만든 외부 함수를 호출하면 함수 호출 전에 *errno* 값이 ctypes 내부 복사본과 스와프되며 함수 호출 직후에도 마찬가지 작업을 합니다.

ctypes.get_errno() 함수는 ctypes 내부 사본의 값을 반환하고, *ctypes.set_errno()* 함수는 ctypes 내부 사본을 새 값으로 변경하고 이전 값을 반환합니다.

use_last_error 매개 변수를 참으로 설정하면, *GetLastError()* 와 `SetLastError()` 윈도우 API 함수에서 관리하는 윈도우 에러 코드에 대해 같은 메커니즘을 활성화합니다. *ctypes.get_last_error()*와 *ctypes.set_last_error()*는 윈도우 에러 코드의 ctypes 내부 사본을 요청하고 변경하는 데 사용됩니다.

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.

공유 라이브러리는 *LibraryLoader* 클래스의 인스턴스인 사전 작성된 객체 중 하나를 사용하여 로드 할 수도 있습니다. 로더의 LoadLibrary() 메서드를 호출하거나 로더 인스턴스의 어트리뷰트로 라이브러리를 조회합니다.

class ctypes.**LibraryLoader**(*dlltype*)
공유 라이브러리를 로드하는 클래스. *dlltype*은 *CDLL*, *PyDLL*, *WinDLL* 또는 *OleDLL* 형 중 하나여야 합니다.

__getattr__()에는 특수한 동작이 있습니다: 라이브러리 로더 인스턴스의 어트리뷰트를 액세스하여 공유 라이브러리를 로드 할 수 있게 합니다. 결과는 캐시 되므로 반복되는 어트리뷰트 액세스는 매번 같은 라이브러리를 반환합니다.

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**
윈도우 전용: *WinDLL* 인스턴스를 만듭니다.

ctypes.**oledll**
윈도우 전용: *OleDLL* 인스턴스를 만듭니다.

ctypes.**pydll**
PyDLL 인스턴스를 만듭니다.

C 파이썬 API에 직접 액세스하기 위해, 바로 사용할 수 있는 파이썬 공유 라이브러리 객체가 제공됩니다:

ctypes.**pythonapi**
파이썬 C API 함수를 어트리뷰트로 노출하는 *PyDLL*의 인스턴스. 이 모든 함수는 C int를 반환한다고 가정하는데, 물론 항상 옳지는 않습니다. 그런 함수를 사용하려면 올바른 restype 어트리뷰트를 대입해야 합니다.

외부 함수

이전 섹션에서 설명한 것처럼, 외부 함수는 로드된 공유 라이브러리의 어트리뷰트로 액세스할 수 있습니다. 이런 방식으로 만들어진 함수 객체는 기본적으로 임의의 개수 인자를 허용하고, 임의의 ctypes 데이터 인스턴스를 인자로 받아들이고, 라이브러리 로더에 의해 지정된 기본 결과형을 반환합니다. 이것들은 내부 클래스의 인스턴스입니다:

class ctypes.**_FuncPtr**
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
ctypes 형을 대입하여 외부 함수의 결과형을 지정합니다. 아무것도 반환하지 않는 함수인 void는 None를 사용하십시오.

ctypes 형이 아닌 콜러블 파이썬 객체를 대입할 수 있습니다. 이때 함수는 C int를 반환한다고 가정하고, 이 콜러블 객체는 이 정수로 호출되어, 사후 처리나 에러 검사를 허용합니다. 이 기능을 사용하

는 것은 폐지되었습니다. 더 유연한 사후 처리나 에러 검사를 위해, ctypes 데이터형을 *restype*로 사용하고, 콜러블을 *errcheck* 어트리뷰트에 대입하십시오.

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.

외부 함수가 호출될 때, 각 실제 인자는 *argtypes* 튜플의 항목의 `from_param()` 클래스 메서드에 전달됩니다. 이 메서드는 실제 인자를 외부 함수가 받아들이는 객체에 맞출 수 있습니다. 예를 들어, *argtypes* 튜플의 *c_char_p* 항목은 ctypes 변환 규칙을 사용하여 인자로 전달된 문자열을 바이트열 객체로 변환합니다.

새로운 기능: 이제 ctypes 형이 아닌 항목을 argtypes에 넣을 수 있습니다. 하지만 각 항목에는 인자로 사용할 수 있는 값(정수, 문자열, ctypes 인스턴스)을 반환하는 `from_param()` 메서드가 있어야 합니다. 이를 통해 사용자 정의 객체를 함수 매개 변수로 맞출 수 있는 어댑터를 정의 할 수 있습니다.

errcheck
이 어트리뷰트에 파이썬 함수나 다른 콜러블을 대입합니다. 콜러블은 3개 이상의 인자로 호출됩니다:

callable(*result*, *func*, *arguments*)
*result*는 `restype` 어트리뷰트에 지정된 대로 외부 함수가 반환하는 것입니다.

*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.

exception ctypes.**ArgumentError**
외부 함수 호출이 전달된 인자 중 하나를 변환할 수 없을 때 발생하는 예외.

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*)
윈도우 전용: 반환된 함수 프로토타입은 `stdcall` 호출 규칙을 사용하는 함수를 만듭니다. 단 *WINFUNCTYPE()*이 *CFUNCTYPE()*과 같은 윈도우 CE는 예외입니다. 이 함수는 호출 중에 GIL을 해제합니다. *use_errno* 와 *use_last_error*는 위에서와 같은 의미가 있습니다.

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.

COM 메서드는 특별한 호출 규칙을 사용합니다: `argtypes` 튜플에 지정된 매개 변수 외에, 첫 번째 인자로 COM 인터페이스에 대한 포인터가 필요합니다.

선택적 *paramflags* 매개 변수는 위에 설명된 기능보다 훨씬 많은 기능을 갖는 외부 함수 래퍼를 만듭니다.

*paramflags*는 `argtypes`와 같은 길이의 튜플이어야 합니다.

이 튜플의 각 항목에는 매개 변수에 대한 추가 정보가 들어 있으며, 한 개, 두 개 또는 세 개의 항목이 들어있는 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.

이 예제는 기본값이 있는 매개 변수와 이름있는 인자를 지원하도록 윈도우 `MessageBoxW` 함수를 래핑하는 방법을 보여줍니다. 윈도우 헤더 파일의 C 선언은 다음과 같습니다:

```
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")
>>> 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.

출력 매개 변수는 errcheck 프로토콜과 결합하여 추가적인 출력 처리와 에러 검사를 수행할 수 있습니다. win32 GetWindowRect api 함수는 성공이나 실패를 알리기 위해 BOOL을 반환하므로, 이 함수는 에러 검사를 수행하고 API 호출이 실패했을 때 예외를 발생시킬 수 있습니다:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

errcheck 함수가 수신한 인자 튜플을 변경 없이 반환하면, *ctypes*는 출력 매개 변수에 수행하는 일반 처리를 계속합니다. RECT 인스턴스 대신 창 좌표의 튜플을 반환하려면, 함수에서 필드를 조회해서 대신 반환하면 됩니다, 이때는 일반 처리가 더는 수행되지 않습니다:

```
>>> 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.

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.**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_or_size*, *size=None*)
이 함수는 가변 문자 버퍼를 만듭니다. 반환된 객체는 *c_char*의 ctypes 배열입니다.

*init_or_size*는 배열의 크기를 지정하는 정수거나 배열 항목을 초기화하는 데 사용될 바이트열 객체여야 합니다.

바이트열 객체가 첫 번째 인자로 지정되면, 버퍼의 길이는 이 객체의 길이보다 한 항목만큼 길어져서, 배열의 마지막 요소가 NUL 종료 문자가 됩니다. 두 번째 인자로 정수를 전달하면 바이트열의 길이를 사용하지 않고 배열의 크기를 지정할 수 있습니다.

ctypes.**create_unicode_buffer**(*init_or_size*, *size=None*)
이 함수는 가변 유니코드 문자 버퍼를 만듭니다. 반환된 객체는 *c_wchar*의 ctypes 배열입니다.

*init_or_size*는 배열의 크기를 지정하는 정수거나 배열 항목을 초기화하는 데 사용될 문자열이어야 합니다.

문자열이 첫 번째 인자로 지정되면, 버퍼의 길이는 문자열의 길이보다 한 항목만큼 길어져서, 배열의 마지막 요소가 NUL 종료 문자가 됩니다. 두 번째 인자로 정수를 전달하면 문자열의 길이를 사용하지 않고 배열의 크기를 지정할 수 있습니다.

ctypes.**DllCanUnloadNow**()
윈도우 전용: 이 함수는 ctypes로 프로세스 내부(in-process) COM 서버를 구현하게 하는 훅입니다. _ctypes 확장 dll이 내보내는 DllCanUnloadNow 함수에서 호출됩니다.

ctypes.**DllGetClassObject**()
윈도우 전용: 이 함수는 ctypes로 프로세스 내부(in-process) COM 서버를 구현하게 하는 훅입니다. `_ctypes` 확장 dll이 내보내는 DllGetClassObject 함수에서 호출됩니다.

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.

ctypes.util.**find_msvcrt**()
윈도우 전용: 파이썬과 확장 모듈이 사용하는 VC 런타임 라이브러리의 파일명을 반환합니다. 라이브러리의 이름을 판별할 수 없으면 `None`이 반환됩니다.

예를 들어, free(void *)에 대한 호출로 확장 모듈에 의해 할당된 메모리를 해제해야 하면, 메모리를 할당한 것과 같은 라이브러리에 있는 함수를 사용하는 것이 중요합니다.

ctypes.**FormatError**([*code*])
윈도우 전용: 에러 코드 *code*의 텍스트 설명을 반환합니다. 에러 코드를 지정하지 않으면 윈도우 API 함수 GetLastError를 호출하여 마지막 에러 코드가 사용됩니다.

ctypes.**GetLastError**()
윈도우 전용: 호출 스레드에서 윈도우가 설정한 마지막 에러 코드를 반환합니다. 이 함수는 윈도우 *GetLastError()* 함수를 직접 호출합니다. 에러 코드의 ctypes 내부 복사본을 반환하지 않습니다.

ctypes.**get_errno**()
호출하는 스레드에서 시스템 *errno* 변수의 ctypes 내부 복사본의 현재 값을 반환합니다.

ctypes.**get_last_error**()
윈도우 전용: 호출 중인 스레드에서 시스템 LastError 변수의 ctypes 내부 복사본의 현재 값을 반환합니다.

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*)
이 팩토리 함수는 새로운 ctypes 포인터형을 만들고 반환합니다. 포인터형은 캐시 되고 내부적으로 재사용되므로, 이 함수를 반복적으로 호출하는 것은 저렴합니다. *type*는 ctypes 형이어야 합니다.

ctypes.**pointer**(*obj*)
이 함수는 *obj*를 가리키는 새 포인터 인스턴스를 만듭니다. 반환된 객체는 형 POINTER(type(obj))입니다.

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.

ctypes.**set_last_error**(*value*)
윈도우 전용: 호출 중인 스레드의 시스템 LastError 변수의 ctypes 내부 복사본의 현재 값을 *value*로 설정하고 이전 값을 반환합니다.

ctypes.**sizeof**(*obj_or_type*)
ctypes 형이나 인스턴스 메모리 버퍼의 크기를 바이트 단위로 반환합니다. C sizeof 연산자와 같은 일을 합니다.

ctypes.**string_at**(*address*, *size=-1*)
이 함수는 메모리 주소 *address*에서 시작하는 C 문자열을 바이트열 객체로 반환합니다. size가 지정되면 크기로 사용되며, 그렇지 않으면 문자열은 0으로 종료된다고 가정합니다.

ctypes.**WinError**(*code=None*, *descr=None*)
윈도우 전용: 이 함수는 아마도 ctypes에서 가장 이름을 잘못 붙인 것입니다. OSError의 인스턴스를 만

듭니다. *code*를 지정하지 않으면, 에러 코드를 판별하기 위해 `GetLastError`가 호출됩니다. *descr*가 지정되지 않으면, 에러에 대한 텍스트 설명을 얻기 위해 *FormatError()*가 호출됩니다.

버전 3.3에서 변경: *WindowsError*의 인스턴스를 만들어왔습니다.

ctypes.**wstring_at**(*address*, *size=-1*)
이 함수는 메모리 주소 *address*에서 시작하는 광폭(wide) 문자열을 문자열로 반환합니다. *size*가 지정되면, 문자열의 문자 수로 사용되며, 그렇지 않으면 문자열은 0으로 종료된다고 가정합니다.

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.

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.

from_address(*address*)
이 메서드는 정수 *address*로 지정된 메모리를 사용하여 ctypes 형 인스턴스를 반환합니다.

from_param(*obj*)
이 메서드는 *obj*를 ctypes 형에게 맞게 조정합니다. 형이 외부 함수의 `argtypes` 튜플에 존재할 때, 외부 함수 호출에 사용된 실제 객체로 호출됩니다; 함수 호출 매개 변수로 사용할 수 있는 객체를 반환 해야 합니다.

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.

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 서브 클래스이므로, 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.

기본 데이터형은, 외부 함수 호출 결과로 반환되거나 (예를 들어) 구조체 필드 멤버나 배열 항목을 꺼낼 때, 원시 (native) 파이썬 형으로 투명하게 변환됩니다. 즉, 외부 함수가 *c_char_p*인 `restype`을 가지면, 항상 *c_char_p* 인스턴스가 아니라 파이썬 바이트열 객체를 받습니다.

기본 데이터형의 서브 클래스는 이 동작을 상속하지 않습니다. 따라서 외부 함수의 `restype`가 *c_void_p*의 서브 클래스면, 함수 호출에서 이 서브 클래스의 인스턴스를 받게 됩니다. 물론, `value` 어트리뷰트에 액세스해서 포인터 값을 가져올 수 있습니다.

다음은 기본 ctypes 데이터형입니다:

class ctypes.**c_byte**
C `signed char` 데이터형을 나타내고, 값을 작은 정수로 해석합니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_char**
C `char` 데이터형을 나타내고, 값을 단일 문자로 해석합니다. 생성자는 선택적 문자열 초기화자를 받아들입니다, 문자열의 길이는 정확히 한 문자여야 합니다.

class ctypes.**c_char_p**
0으로 끝나는 문자열을 가리킬 때, C `char *` 데이터형을 나타냅니다. 바이너리 데이터를 가리킬 수도 있는 일반 문자 포인터를 위해서는, `POINTER(c_char)`를 사용해야 합니다. 생성자는 정수 주소나 바이트열 객체를 받아들입니다.

class ctypes.**c_double**
C `double` 데이터형을 나타냅니다. 생성자는 선택적 float 초기화자를 받아들입니다.

class ctypes.**c_longdouble**
C `long double` 데이터형을 나타냅니다. 생성자는 선택적 float 초기화자를 받아들입니다. `sizeof(long double) == sizeof(double)`인 플랫폼에서 *c_double*의 별칭입니다.

class ctypes.**c_float**
C `float` 데이터형을 나타냅니다. 생성자는 선택적 float 초기화자를 받아들입니다.

class ctypes.**c_int**
C `signed int` 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다. 오버플로 검사는 수행되지 않습니다. `sizeof(int) == sizeof(long)`인 플랫폼에서 *c_long*의 별칭입니다.

class ctypes.**c_int8**
C 8비트 `signed int` 데이터형을 나타냅니다. 보통 *c_byte*의 별칭입니다.

class ctypes.**c_int16**
C 16비트 `signed int` 데이터형을 나타냅니다. 보통 *c_short*의 별칭입니다.

class ctypes.**c_int32**
C 32비트 signed int 데이터형을 나타냅니다. 보통 *c_int*의 별칭입니다.

class ctypes.**c_int64**
C 64비트 signed int 데이터형을 나타냅니다. 보통 *c_longlong*의 별칭입니다.

class ctypes.**c_long**
C signed long 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_longlong**
C signed long long 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_short**
C signed short 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_size_t**
C size_t 데이터형을 나타냅니다.

class ctypes.**c_ssize_t**
C ssize_t 데이터형을 나타냅니다.

버전 3.2에 추가.

class ctypes.**c_ubyte**
C unsigned char 데이터형을 나타내고, 값을 작은 정수로 해석합니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_uint**
C unsigned int 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다. sizeof(int) == sizeof(long)인 플랫폼에서 *c_ulong*의 별칭입니다.

class ctypes.**c_uint8**
C 8비트 unsigned int 데이터형을 나타냅니다. 보통 *c_ubyte*의 별칭입니다.

class ctypes.**c_uint16**
C 16비트 unsigned int 데이터형을 나타냅니다. 보통 *c_ushort*의 별칭입니다.

class ctypes.**c_uint32**
C 32비트 unsigned int 데이터형을 나타냅니다. 보통 *c_uint*의 별칭입니다.

class ctypes.**c_uint64**
C 64비트 unsigned int 데이터형을 나타냅니다. 보통 *c_ulonglong*의 별칭입니다.

class ctypes.**c_ulong**
C unsigned long 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_ulonglong**
C unsigned long long 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_ushort**
C unsigned short 데이터형을 나타냅니다. 생성자는 선택적 정수 초기화자를 받아들입니다; 오버플로 검사는 수행되지 않습니다.

class ctypes.**c_void_p**
C void * 형을 나타냅니다. 값은 정수로 표시됩니다. 생성자는 선택적 정수 초기화자를 받아들입니다.

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**
0으로 끝나는 광폭 문자 문자열을 가리키는 포인터여야 하는 C wchar_t * 데이터형을 나타냅니다. 생성자는 정수 주소나 문자열을 받아들입니다.

class ctypes.**c_bool**
C bool 데이터형을 나타냅니다 (더욱 정확하게, C99의 _Bool). 이 값은 True나 False 일 수 있고, 생성자는 논릿값이 있는 임의의 객체를 받아들입니다.

class ctypes.**HRESULT**
윈도우 전용: 함수 또는 메서드 호출에 대한 성공 또는 에러 정보가 들어있는, HRESULT 값을 나타냅니다.

class ctypes.**py_object**
C PyObject * 데이터형을 나타냅니다. 인자 없이 이것을 호출하면 NULL PyObject * 포인터가 만들어집니다.

ctypes.wintypes 모듈은 다른 윈도우 특정 데이터형을 제공합니다, 예를 들어, HWND, WPARAM 또는 DWORD. MSG나 RECT와 같은 유용한 구조체도 정의됩니다.

구조화된 데이터형

class ctypes.**Union**(**args*, ***kw*)
네이티브 바이트 순서의 공용체를 위한 추상 베이스 클래스.

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.

네이티브가 아닌 바이트 순서를 갖는 구조체는 포인터형 필드나 포인터형 필드를 포함하는 다른 데이터형을 포함할 수 없습니다.

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)),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                ...
            ]
```

하지만, 형이 처음 사용되기 전에 *_fields_* 클래스 변수를 정의해야 합니다 (인스턴스가 만들어지고, *sizeof()*가 호출되는 등의 일이 일어납니다). 나중에 *_fields_* 클래스 변수에 대입하면 AttributeError가 발생합니다.

구조체 형의 서브-서브 클래스를 정의할 수 있습니다. 베이스 클래스의 필드를 상속하고, 여기에 서브-서브 클래스에 정의된 *_fields_*의 필드가 추가됩니다.

pack

인스턴스의 구조체 필드 정렬을 재정의할 수 있는 선택적 작은 정수입니다. *_fields_*가 대입될 때 *_pack_*는 이미 정의되어 있어야 합니다. 그렇지 않으면 아무 효과가 없습니다.

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.lptdesc와 td.u.lptdesc는 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.

배열과 포인터

class ctypes.**Array**(**args*)
배열의 추상 베이스 클래스.

구상 배열형을 만드는 데 권장되는 방법은, 임의의 *ctypes* 데이터형에 양의 정수를 곱하는 것입니다. 또는, 이 형의 서브 클래스를 만들고, *_length_* 와 *_type_* 클래스 변수를 정의할 수 있습니다. 배열 요소는 표준 서브 스크립트나 슬라이스 액세스를 사용해서 읽고 쓸 수 있습니다; 슬라이스 읽기의 경우, 결과 객체는 *Array*가 아닙니다.

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.

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.

CHAPTER 17

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:

17.1 `threading` — Thread-based parallelism

Source code: Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level *`_thread`* module. See also the *`queue`* module.

버전 3.7에서 변경: This module used to be optional, it is now always available.

참고: While they are not listed below, the `camelCase` names used for some methods and functions in this module in the Python 2.x series are still supported by this module.

CPython implementation detail: In CPython, due to the *Global Interpreter Lock*, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use *`multiprocessing`* or *`concurrent.futures.ProcessPoolExecutor`*. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

This module defines the following functions:

`threading.`**`active_count`**`()`
Return the number of *`Thread`* objects currently alive. The returned count is equal to the length of the list returned by *`enumerate()`*.

`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.

threading.**get_ident**()
Return the 'thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

버전 3.3에 추가.

threading.**enumerate**()
Return a list of all *Thread* objects currently alive. The list includes daemonic threads, dummy thread objects created by *current_thread()*, and the main thread. It excludes terminated threads and threads that have not yet been started.

threading.**main_thread**()
Return the main *Thread* object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

버전 3.4에 추가.

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.**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.**stack_size**([*size*])
Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a *RuntimeError* is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

Availability: Windows, systems with POSIX threads.

This module also defines the following constant:

threading.**TIMEOUT_MAX**
The maximum value allowed for the *timeout* parameter of blocking functions (*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()*, etc.). Specifying a timeout greater than this value will raise an *OverflowError*.

버전 3.2에 추가.

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python's *Thread* class supports a subset of the behavior of Java's Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java's Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

17.1.1 Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of *local* (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

class threading.**local**
A class that represents thread-local data.

For more details and extensive examples, see the documentation string of the _threading_local module.

17.1.2 Thread Objects

The *Thread* class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the *run()* method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the __init__() and *run()* methods of this class.

Once a thread object is created, its activity must be started by calling the thread's *start()* method. This invokes the *run()* method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its *run()* method terminates – either normally, or by raising an unhandled exception. The *is_alive()* method tests whether the thread is alive.

Other threads can call a thread's *join()* method. This blocks the calling thread until the thread whose *join()* method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the *name* attribute.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the *daemon* property or the *daemon* constructor argument.

참고: Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an *Event*.

There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads", which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be *join()*ed. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the *run()* method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-*N*" where *N* is a small decimal number.

args is the argument tuple for the target invocation. Defaults to `()`.

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If not `None`, *daemon* explicitly sets whether the thread is daemonic. If `None` (the default), the daemonic property is inherited from the current thread.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

버전 3.3에서 변경: Added the *daemon* argument.

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's *run()* method to be invoked in a separate thread of control.

This method will raise a *RuntimeError* if called more than once on the same thread object.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the *target* argument, if any, with positional and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

join(*timeout=None*)
Wait until the thread terminates. This blocks the calling thread until the thread whose *join()* method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As *join()* always returns `None`, you must call *is_alive()* after *join()* to decide whether a timeout happened – if the thread is still alive, the *join()* call timed out.

When the *timeout* argument is not present or `None`, the operation will block until the thread terminates.

A thread can be *join()*ed many times.

join() raises a *RuntimeError* if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to *join()* a thread before it has been started and attempts to do so raise the same exception.

name
A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

getName()
setName()
Old getter/setter API for *name*; use it directly as a property instead.

ident
The 'thread identifier' of this thread or `None` if the thread has not been started. This is a nonzero integer. See the *get_ident()* function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

is_alive()
Return whether the thread is alive.

This method returns `True` just before the *run()* method starts until just after the *run()* method terminates. The module function *enumerate()* returns a list of all alive threads.

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before *start()* is called, otherwise *RuntimeError* is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to *daemon* = `False`.

The entire Python program exits when no alive non-daemon threads are left.

isDaemon()
setDaemon()
Old getter/setter API for *daemon*; use it directly as a property instead.

17.1.3 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the *_thread* extension module.

A primitive lock is in one of two states, "locked" or "unlocked". It is created in the unlocked state. It has two basic methods, *acquire()* and *release()*. When the state is unlocked, *acquire()* changes the state to locked and returns immediately. When the state is locked, *acquire()* blocks until a call to *release()* in another thread changes it to unlocked, then the *acquire()* call resets it to locked and returns. The *release()* method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a *RuntimeError* will be raised.

Locks also support the *context management protocol*.

When more than one thread is blocked in *acquire()* waiting for the state to turn to unlocked, only one thread proceeds when a *release()* call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.**Lock**
The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

Note that `Lock` is actually a factory function which returns an instance of the most efficient version of the concrete Lock class that is supported by the platform.

acquire(*blocking=True*, *timeout=-1*)
Acquire a lock, blocking or non-blocking.

When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately; otherwise, set the lock to locked and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is false.

The return value is `True` if the lock is acquired successfully, `False` if not (for example if the *timeout* expired).

버전 3.2에서 변경: The *timeout* parameter is new.

버전 3.2에서 변경: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

release()
Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a *RuntimeError* is raised.

There is no return value.

locked()
Return true if the lock is acquired.

17.1.4 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of "owning thread" and "recursion level" in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its *acquire()* method; this returns once the thread owns the lock. To unlock the lock, a thread calls its *release()* method. *acquire()*/*release()* call pairs may be nested; only the final *release()* (the *release()* of the outermost pair) resets the lock to unlocked and allows another thread blocked in *acquire()* to proceed.

Reentrant locks also support the *context management protocol*.

class threading.**RLock**
This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Note that RLock is actually a factory function which returns an instance of the most efficient version of the concrete RLock class that is supported by the platform.

acquire(*blocking=True*, *timeout=-1*)
Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the *blocking* argument set to true, do the same thing as when called without arguments, and return True.

When invoked with the *blocking* argument set to false, do not block. If a call without an argument would block, return False immediately; otherwise, do the same thing as when called without arguments, and return True.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. Return True if the lock has been acquired, false if the timeout has elapsed.

버전 3.2에서 변경: The *timeout* parameter is new.

release()
Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked

(not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A *RuntimeError* is raised if this method is called when the lock is unlocked.

There is no return value.

17.1.5 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.

A condition variable obeys the *context management protocol*: using the `with` statement acquires the associated lock for the duration of the enclosed block. The *acquire()* and *release()* methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The *wait()* method releases the lock, and then blocks until another thread awakens it by calling *notify()* or *notify_all()*. Once awakened, *wait()* re-acquires the lock and returns. It is also possible to specify a timeout.

The *notify()* method wakes up one of the threads waiting for the condition variable, if any are waiting. The *notify_all()* method wakes up all threads waiting for the condition variable.

Note: the *notify()* and *notify_all()* methods don't release the lock; this means that the thread or threads awakened will not return from their *wait()* call immediately, but only when the thread that called *notify()* or *notify_all()* finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call *wait()* repeatedly until they see the desired state, while threads that modify the state call *notify()* or *notify_all()* when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application's condition is necessary because *wait()* can return after an arbitrary long time, and the condition which prompted the *notify()* call may no longer hold true. This is inherent to multi-threaded programming. The *wait_for()* method can be used to automate the condition checking, and eases the computation of timeouts:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between *notify()* and *notify_all()*, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.**Condition**(*lock=None*)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the *lock* argument is given and not None, it must be a *Lock* or *RLock* object, and it is used as the underlying lock. Otherwise, a new *RLock* object is created and used as the underlying lock.

버전 3.3에서 변경: changed from a factory function to a class.

acquire(**args*)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait(*timeout=None*)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an *RLock*, it is not released using its *release()* method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the *RLock* class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is True unless a given *timeout* expired, in which case it is False.

버전 3.2에서 변경: Previously, the method always returned None.

wait_for(*predicate*, *timeout=None*)

Wait until a condition evaluates to true. *predicate* should be a callable which result will be interpreted as a boolean value. A *timeout* may be provided giving the maximum time to wait.

This utility method may call *wait()* repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to False if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with *wait()*: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

버전 3.2에 추가.

notify(*n=1*)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method wakes up at most *n* of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly *n* threads, if at least *n* threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* threads.

Note: an awakened thread does not actually return from its *wait()* call until it can reacquire the lock. Since *notify()* does not release the lock, its caller should.

notify_all()
Wake up all threads waiting on this condition. This method acts like *notify()*, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

17.1.6 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of *acquire()* and *release()*).

A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. The counter can never go below zero; when *acquire()* finds that it is zero, it blocks, waiting until some other thread calls *release()*.

Semaphores also support the *context management protocol*.

class threading.**Semaphore**(*value=1*)
This class implements semaphore objects. A semaphore manages an atomic counter representing the number of *release()* calls minus the number of *acquire()* calls, plus an initial value. The *acquire()* method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter; it defaults to `1`. If the *value* given is less than 0, *ValueError* is raised.

버전 3.3에서 변경: changed from a factory function to a class.

acquire(*blocking=True*, *timeout=None*)
Acquire a semaphore.

When invoked without arguments:

- If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately.
- If the internal counter is zero on entry, block until awoken by a call to *release()*. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return `True`. Exactly one thread will be awoken by each call to *release()*. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to false, do not block. If a call without an argument would block, return `False` immediately; otherwise, do the same thing as when called without arguments, and return `True`.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return `False`. Return `True` otherwise.

버전 3.2에서 변경: The *timeout* parameter is new.

release()
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

class threading.**BoundedSemaphore**(*value=1*)
Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value

doesn't exceed its initial value. If it does, *ValueError* is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

버전 3.3에서 변경: changed from a factory function to a class.

Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

17.1.7 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true.

class threading.**Event**

Class implementing event objects. An event manages a flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true. The flag is initially false.

버전 3.3에서 변경: changed from a factory function to a class.

is_set()

Return `True` if and only if the internal flag is true.

set()

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call *wait()* once the flag is true will not block at all.

clear()

Reset the internal flag to false. Subsequently, threads calling *wait()* will block until *set()* is called to set the internal flag to true again.

wait(*timeout=None*)

Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls *set()* to set the flag to true, or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

This method returns `True` if and only if the internal flag has been set to true, either before the wait call or after the wait starts, so it will always return `True` except if a timeout is given and the operation times out.

버전 3.1에서 변경: Previously, the method always returned `None`.

17.1.8 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. *Timer* is a subclass of *Thread* and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the *cancel()* method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

버전 3.3에서 변경: changed from a factory function to a class.

cancel()

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

17.1.9 Barrier Objects

버전 3.2에 추가.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the *wait()* method and will block until all of the threads have made their *wait()* calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the *wait()* method.

wait(*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* – 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a *BrokenBarrierError* exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the *BrokenBarrierError* exception.

Note that using this function may can require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

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 needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

A boolean that is `True` if the barrier is in the broken state.

exception threading.**BrokenBarrierError**

This exception, a subclass of *RuntimeError*, is raised when the *Barrier* object is reset or broken.

17.1.10 Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire()` and `release()` methods can be used as context managers for a `with` statement. The `acquire()` method will be called when the block is entered, and `release()` will be called when the block is exited. Hence, the following snippet:

```
with some_lock:
    # do something...
```

is equivalent to:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, *Lock*, *RLock*, *Condition*, *Semaphore*, and *BoundedSemaphore* objects may be used as `with` statement context managers.

17.2 multiprocessing — 프로세스 기반 병렬 처리

소스 코드: Lib/multiprocessing/

17.2.1 소개

multiprocessing 은 *threading* 모듈과 유사한 API를 사용하여 프로세스 스포닝(spawning)을 지원하는 패키지입니다. *multiprocessing* 패키지는 지역과 원격 동시성을 모두 제공하며 스레드 대신 서브 프로세스를 사용하여 전역 인터프리터 록 을 효과적으로 피합니다. 이것 때문에, *multiprocessing* 모듈은 프로그래머가 주어진 기계에서 다중 프로세서를 최대한 활용할 수 있게 합니다. 유닉스와 윈도우에서 모두 실행됩니다.

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]
```

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.

컨텍스트 및 시작 방법

플랫폼에 따라, *multiprocessing*은 프로세스를 시작하는 세 가지 방법을 지원합니다. 이러한 시작 방법 은

spawn 부모 프로세스는 깨끗한 새 파이썬 인터프리터 프로세스를 시작합니다. 자식 프로세스는 프로세스 객체의 *run()* 메서드를 실행하는데 필요한 자원만 상속받습니다. 특히, 부모 프로세스의 불필요한 파일 기술자와 핸들은 상속되지 않습니다. 이 방법을 사용하여 프로세스를 시작하는 것은 *fork* 나 *forkserver* 를 사용하는 것에 비해 다소 느립니다.

유닉스 및 윈도우에서 사용 가능합니다. 윈도우의 기본값.

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.

유닉스에서만 사용 가능합니다. 유닉스의 기본값.

forkserver 프로그램이 시작되고 *forkserver* 시작 방법을 선택하면, 서버 프로세스가 시작됩니다. 그 이후부터, 새로운 프로세스가 필요할 때마다, 부모 프로세스는 서버에 연결하여 새로운

프로세스를 포크 하도록 요청합니다. 포크 서버 프로세스는 단일 스레드이므로 *os.fork()* 를 사용하는 것이 안전합니다. 불필요한 자원은 상속되지 않습니다.

유닉스 파이프를 통해 파일 기술자를 전달할 수 있는 유닉스 플랫폼에서 사용할 수 있습니다.

버전 3.4에서 변경: 모든 유닉스 플랫폼에 *spawn* 이 추가되었고, 일부 유닉스 플랫폼에는 *forkserver* 가 추가되었습니다. 윈도우에서 자식 프로세스는 상속 가능한 모든 부모 핸들을 더는 상속하지 않습니다.

유닉스에서 *spawn* 또는 *forkserver* 시작 방법을 사용하면 세마포어 추적기 프로세스 역시 시작되는데, 프로그램의 프로세스들이 만든 삭제되지 않은 이름있는 세마포어를 추적합니다. 모든 프로세스가 종료된 후 세마포어 추적기는 남아있는 세마포어를 제거합니다. 일반적으로 아무것도 남아 있지 않아야 하지만, 프로세스가 시그널에 의해 죽으면 "누수된" 세마포어가 있을 수 있습니다. (이름있는 세마포어의 제거는 심각한 문제인데, 시스템이 제한된 수만 허용하고 다음 재부팅 때까지 자동으로 제거되지 않기 때문입니다.)

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')

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.

특정 시작 방법을 사용하고자 하는 라이브러리는 아마도 *get_context()* 를 사용하여 라이브러리 사용자의 선택을 방해하지 않아야 합니다.

경고: `'spawn'` 과 `'forkserver'` 시작 방법은 현재 유닉스에서 "고정된(frozen)" 실행 파일(즉, **PyInstaller**와 **cx_Freeze**와 같은 패키지로 만든 바이너리)과 함께 사용할 수 없습니다. `'fork'` 시작 방법은 작동합니다.

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()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

큐는 스레드와 프로세스에 안전합니다.

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.

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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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.

Manager() 가 반환한 관리자는 *list*, *dict*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value* 그리고 *Array* 형을 지원합니다. 예를 들어, 다음 코드는

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

를 인쇄할 것입니다

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        # 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 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        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):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(이것을 시도해 보면 실제로 세 개의 전체 트레이스백이 어느 정도 임의로 번갈아 출력됩니다. 그런 다음 마스터 프로세스를 중지시켜야 할 수도 있습니다.)

17.2.2 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.

생성자는 항상 키워드 인자로 호출해야 합니다. *group* 은 항상 None 이어야 합니다; 이것은 *threading.Thread* 와의 호환성을 위해서만 존재합니다. *target* 은 *run()* 메서드에 의해 호출될 콜러블 객체입니다. 기본값은 None 인데, 아무것도 호출되지 않음을 의미합니다. *name* 은 프로세스 이름입니다 (자세한 내용은 *name* 참조). *args* 는 target 호출을 위한 인자 튜플입니다. *kwargs* 는 target 호출을 위한 키워드 인자 딕셔너리입니다. 제공되는 경우, 키워드 전용 *daemon* 인자는 프로세스 *daemon* 플래그를 True 또는 False 로 설정합니다. None (기본값) 이면, 이 플래그는 만드는 프로세스로부터 상속됩니다.

기본적으로, 아무 인자도 *target* 에 전달되지 않습니다.

서브 클래스가 생성자를 재정의하면, 프로세스에 다른 작업을 하기 전에 베이스 클래스 생성자 (Process.__init__())를 호출해야 합니다.

버전 3.3에서 변경: *daemon* 인자가 추가되었습니다.

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.

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
자식의 종료 코드. 프로세스가 아직 종료되지 않았으면 None 이 됩니다. 음수 값 *-N* 은 자식이 시그널 *N* 에 의해 강제 종료되었음을 나타냅니다.

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.

multiprocessing.connection.wait() 를 사용해서 한 번에 여러 이벤트를 기다리고 싶다면, 이 값을 사용할 수 있습니다. 그렇지 않으면 *join()* 을 호출하는 것이 더 간단합니다.

윈도우에서, 이것은 WaitForSingleObject 및 WaitForMultipleObjects 계열의 API 호출에서 사용할 수 있는 OS 핸들입니다. 유닉스에서, 이것은 *select* 모듈의 프리미티브들에서 사용할 수 있는 파일 기술자입니다.

버전 3.3에 추가.

terminate()
프로세스를 강제 종료합니다. 유닉스에서는 SIGTERM 시그널을 사용합니다; 윈도우에서는 TerminateProcess() 가 사용됩니다. 종료 처리기(exit handler)와 finally 절 등이 실행되지 않음에 주의하십시오.

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()
terminate() 와 같지만, 유닉스에서 SIGKILL 시그널을 사용합니다.

버전 3.7에 추가.

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.

버전 3.7에 추가.

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
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**
모든 *multiprocessing* 예외의 베이스 클래스입니다.

exception multiprocessing.**BufferTooShort**
Connection.recv_bytes_into() 가, 제공된 버퍼 객체가 읽은 메시지에 비해 너무 작을 때 일으키는 예외.

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.

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.

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.

이것은 sem_getvalue() 가 구현되지 않은 Mac OS X와 같은 유닉스 플랫폼에서 *NotImplementedError*를 발생시킬 수 있습니다.

empty()

큐가 비어 있다면 True 를, 그렇지 않으면 False 를 반환합니다. 다중 스레딩/다중 프로세싱 특성을

타기 때문에 신뢰할 수 없습니다.

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).

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 (block이 False) 즉시 사용할 수 있는 항목이 있으면 반환하고, 그렇지 않으면 *queue.Empty* 예외를 발생시킵니다 (이 경우 *timeout* 은 무시됩니다).

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()
현재 프로세스가 이 큐에 더는 데이터를 넣지 않을 것을 나타냅니다. 버퍼에 저장된 모든 데이터를 파이프로 플러시 하면 배경 스레드가 종료됩니다. 큐가 가비지 수집될 때 자동으로 호출됩니다.

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.

이 메서드의 더 좋은 이름은 allow_exit_without_flush() 일 것입니다. 큐에 포함된 데이터가 유실될 가능성이 크며, 거의 확실히 사용할 필요가 없을 겁니다. 현재 프로세스가 하부 파이프로 대기 중인 데이터를 플러시 할 때까지 기다리지 않고 즉시 종료해야 하고 데이터 손실에 대해서는 신경 쓰지 않을 때만을 위한 것입니다.

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.

empty()
큐가 비어 있다면 True 를, 그렇지 않으면 False 를 반환합니다.

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.

이 숫자는 현재 프로세스에서 사용할 수 있는 CPU 수와 같지 않습니다. 사용 가능한 CPU 수는 `len(os.sched_getaffinity(0))` 로 얻을 수 있습니다.

*NotImplementedError*를 일으킬 수 있습니다.

더 보기:

os.cpu_count()

multiprocessing.**current_process**()
현재 프로세스에 해당하는 *Process* 객체를 반환합니다.

threading.current_thread() 와 유사한 기능을 제공합니다.

multiprocessing.**freeze_support**()
*multiprocessing*을 사용하는 프로그램이 고정되어(frozen) 윈도우 실행 파일을 생성할 때를 위한 지원을 추가합니다. (**py2exe**, **PyInstaller** 및 **cx_Freeze** 에서 테스트 되었습니다.)

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!')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

freeze_support() 줄이 생략된 경우 고정된 실행 파일을 실행하려고 하면 *RuntimeError* 가 발생합니다.

freeze_support() 호출은 윈도우가 아닌 다른 운영 체제에서 실행될 때는 아무런 영향을 미치지 않습니다. 또한, 모듈이 윈도우상의 파이썬 인터프리터에 의해 정상적으로 실행되는 경우 (프로그램이 고정되지 않은 경우)에도 freeze_support() 는 아무 효과가 없습니다.

multiprocessing.**get_all_start_methods**()
지원되는 시작 방법의 리스트를 반환하는데, 그 중 첫 번째가 기본값입니다. 가능한 시작 방법은 'fork', 'spawn' 및 'forkserver' 입니다. 윈도우에서는 'spawn' 만 사용할 수 있습니다. 유닉스에서는 'fork' 와 'spawn' 이 항상 지원되며 'fork' 가 기본값입니다.

버전 3.4에 추가.

multiprocessing.**get_context**(*method=None*)
multiprocessing 모듈과 같은 어트리뷰트를 가진 컨텍스트 객체를 반환합니다.

method 가 None 이면 기본 컨텍스트가 반환됩니다. 그렇지 않으면 *method* 는 'fork', 'spawn', 'forkserver' 이어야 합니다. 지정된 시작 방법을 사용할 수 없는 경우 *ValueError* 가 발생합니다.

버전 3.4에 추가.

multiprocessing.**get_start_method**(*allow_none=False*)
프로세스를 기동하기 위해서 사용되는 시작 방법의 이름을 돌려줍니다.

시작 방법이 고정되지 않았고 *allow_none* 이 거짓이면, 시작 방법이 기본값으로 고정되고 이름이 반환됩니다. 시작 방법이 고정되지 않았고 *allow_none*이 참이면, None 이 반환됩니다.

반환 값은 'fork', 'spawn', 'forkserver' 또는 None 입니다. 유닉스에서는 'fork' 가 기본값이고, 윈도우에서는 'spawn' 이 기본값입니다.

버전 3.4에 추가.

multiprocessing.**set_executable**()
자식 프로세스를 시작할 때 사용할 파이썬 인터프리터의 경로를 설정합니다. (기본적으로 *sys.executable* 이 사용됩니다). 파이썬은 내장하는 사람들은 아마도 다음과 같이 할 필요가 있습니다

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

자식 프로세스를 만들기 전에 해야 합니다.

버전 3.4에서 변경: 이제 'spawn' 시작 방법을 사용할 때 유닉스에서 지원됩니다.

multiprocessing.**set_start_method**(*method*)
자식 프로세스를 시작하는 데 사용해야 하는 방법을 설정합니다. *method* 는 'fork', 'spawn' 또는 'forkserver' 일 수 있습니다.

이것은 한 번만 호출해야 하며, 메인 모듈의 if __name__ == '__main__' 절 내에서 보호되어야 합니다.

버전 3.4에 추가.

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.

버전 3.3에 추가: 이제 연결 객체는 컨텍스트 관리 프로토콜을 지원합니다 – 컨텍스트 관리자 형를 보세요. *__enter__()* 는 연결 객체를 반환하고, *__exit__()* 는 *close()* 를 호출합니다.

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.

버전 3.3에 추가.

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: Mac OS X에서, `sem_getvalue()` 가 해당 플랫폼에 구현되어 있지 않기 때문에 *Semaphore*와

구별되지 않습니다.

---

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.

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.

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: Mac OS X에서, `sem_timedwait` 가 지원되지 않기 때문에, `acquire()` 를 시간제한 있게 호출하면 잠자는 루프를 사용하여 해당 함수의 동작을 흉내 냅니다.

참고: 메인 스레드가 `BoundedSemaphore.acquire()`, *Lock.acquire()*, *RLock.acquire()*, `Semaphore.acquire()`, `Condition.acquire()` 또는 `Condition.wait()` 호출 때문에 블록 된 동안, `Ctrl-C` 에 의해 만들어진 SIGINT 시그널이 도착하면, 호출이 즉시 중단되고 *KeyboardInterrupt* 가 발생합니다.

이것은 *threading* 의 동작과는 다른데, SIGINT는 해당 블로킹 호출이 진행되는 동안 무시됩니다.

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 동기화 된 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 typecode입니다. **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 동기화 된 래퍼입니다.

typecode_or_type 은 반환된 배열의 요소의 형을 결정합니다: ctypes 형이거나 *array* 모듈에 의해 사용되는 종류의 한 문자 typecode입니다. *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 typecode입니다. *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 typecode입니다. **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*[, *authkey*]])
: 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.

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()
Manager의 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.

버전 3.3에 추가.

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.

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:

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> 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(Worker, self).__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 대상이라고 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:

```
>>> from multiprocessing import Manager
>>> manager = 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 는 사용할 작업자 프로세스 수입니다. *processes* 가 None 이면 *os.cpu_count()* 에 의해 반환되는 수가 사용됩니다.

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* objects have internal resources that need to be properly managed (like any other resource) by using the pool as a context manager or by calling *close()* and *terminate()* manually. Failure to do this can lead to the process hanging on finalization.
>
> Note that is **not correct** to rely on the garbage colletor 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 3.2에 추가: *maxtasksperchild*

버전 3.4에 추가: *context*

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 스폰 되어 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*]]]])
결과 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*])
A parallel equivalent of the `map()` built-in function (it supports only one *iterable* argument though, for multiple iterables see `starmap()`). It blocks until the result is ready.

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*]]])
결과 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*])
`map()` 과 같지만, *iterable* 의 요소가 인자로 언팩 될 이터러블일 것으로 기대합니다.

따라서 *iterable* 이 `[(1,2), (3, 4)]` 이면 결과는 `[func(1,2), func(3,4)]` 가 됩니다.

버전 3.3에 추가.

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.

버전 3.3에 추가.

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 3.3에 추가: 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()
Return whether the call completed without raising an exception. Will raise *ValueError* if the result is not ready.

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)

authkey 가 주어지고 None이 아니라면, 바이트열이어야 하며 HMAC 기반 인증 챌린지의 비밀 키로 사용됩니다. *authkey* 가 None이면, 인증이 수행되지 않습니다. 인증이 실패하면 *AuthenticationError* 가 발생합니다. 인증 키를 보세요.

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.

authkey 가 주어지고 None이 아니라면, 바이트열이어야 하며 HMAC 기반 인증 챌린지의 비밀 키로 사용됩니다. *authkey* 가 None이면, 인증이 수행되지 않습니다. 인증이 실패하면 *AuthenticationError* 가 발생합니다. 인증 키를 보세요.

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 3.3에 추가: 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 float면, 호출이 최대 지정된 초만큼 블록 됩니다. *timeout* 이 None 이면, 시간제한 없이 블록 됩니다. 음수 timeout은 0과 같습니다.

유닉스와 윈도우에서 모두, *object_list* 에 등장할 수 있는 객체는 다음과 같습니다.

- 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.

유닉스: `wait(object_list, timeout)` 은 `select.select(object_list, [], [], timeout)`과 거의 동등합니다. 차이점은, *select.select()* 가 시그널에 의해 인터럽트 되면, 에러 번호 EINTR 로 *OSError* 를 일으키지만, *wait()* 는 예외를 일으키지 않는다는 것입니다.

윈도우: *object_list* 의 항목은 (Win32 함수 `WaitForMultipleObjects()` 의 설명서에서 사용된 정의에 따라) 대기 가능한 정수 핸들이거나, 소켓 핸들이나 파이프 핸들을 반환하는 `fileno()` 메서드가 있는 개체입니다. (파이프 핸들과 소켓 핸들은 대기 가능한 핸들이 **아님** 에 유의하십시오.)

버전 3.3에 추가.

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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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:

```
import time, random
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 = []

    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.
- **`'AF_PIPE'` 주소는 형식** `r'\.\pipe{PipeName}'` 의 문자열입니다. *Client()* 를 사용하여 *ServerName* 이라는 원격 컴퓨터의 이름있는 파이프에 연결하려면, 대신 `r'\`*ServerName*`\pipe{PipeName}'` 형식의 주소를 사용해야 합니다.

두 개의 역 슬래시로 시작하는 문자열은 기본적으로 `'AF_UNIX'` 주소가 아니라 `'AF_PIPE'` 주소로 간주합

니다.

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.

로거가 처음 생성되면 수준 `logging.NOTSET` 을 가지며 기본 처리기가 없습니다. 이 로거로 보낸 메시지는 기본적으로 루트 로거에 전파되지 않습니다.

윈도우에서 자식 프로세스는 부모 프로세스의 로거의 수준만 상속받습니다 – 그 밖의 다른 로거 사용자 지정은 상속되지 않습니다.

`multiprocessing.`**`log_to_stderr`**`()`
이 함수는 *get_logger()*를 호출하지만, get_logger에 의해 생성된 로거를 반환하는 것 외에, `'[%(levelname)s/%(processName)s] %(message)s'` 포맷을 사용하여 *sys.stderr* 에 출력을 전송하는 처리기를 추가합니다.

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.

In particular, the `Pool` function provided by *multiprocessing.dummy* returns an instance of *ThreadPool*, which is a subclass of *Pool* that supports all the same method calls but uses a pool of worker threads rather than worker processes.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])

A thread pool object which controls a pool of worker threads to which jobs can be submitted. *ThreadPool* instances are fully interface compatible with *Pool* instances, and their resources must also be properly managed, either by using the pool as a context manager or by calling *close()* and *terminate()* manually.

processes is the number of worker threads to use. If *processes* is `None` then the number returned by *os.cpu_count()* is used.

initializer 가 `None` 이 아니면, 각 작업자 프로세스는 시작할 때 `initializer(*initargs)` 를 호출합니다.

Unlike *Pool*, *maxtasksperchild* and *context* cannot be provided.

> **참고:** A *ThreadPool* shares the same interface as *Pool*, which is designed around a pool of processes and predates the introduction of the *concurrent.futures* module. As such, it inherits some operations that don't make sense for a pool backed by threads, and it has its own type for representing the status of asynchronous jobs, *AsyncResult*, that is not understood by any other libraries.
>
> Users should generally prefer to use *concurrent.futures.ThreadPoolExecutor*, which has a simpler interface that was designed around threads from the start, and which returns *concurrent.futures.Future* instances that are compatible with many other libraries, including *asyncio*.

17.2.3 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

유닉스에서 프로세스가 끝났지만 조인되지 않으면 좀비가 됩니다. 너무 많이 생기지는 않아야 하는데, 새로운 프로세스가 시작될 때마다 (또는 *active_children()* 이 호출 되면) 아직 조인되지 않은 모든 완료된 프로세스를 조인하기 때문입니다. 또한, 완료된 프로세스의 *Process.is_alive* 를 호출하면 조인합니다. 그렇다고 하더라도 여러분이 시작시키는 모든 프로세스를 명시적으로 조인하는 것이 좋습니다.

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__':
    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.

fork 시작 방법을 사용하는 유닉스에서, 자식 프로세스는 전역 자원을 사용하여 부모 프로세스에서 생성된 공유 자원을 사용할 수 있습니다. 그러나 자식 프로세스의 생성자에 객체를 인자로 전달하는 것이 더 좋습니다.

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)
```

이것은 프로세스가 서로 충돌해서 파일 기술자 에러를 일으키는 근본적인 문제를 해결하지만, *sys.stdin()* 을 출력 버퍼링을 사용하는 "파일과 유사한 객체"로 교체하는 응용 프로그램에 잠재적 위험을 만듭니다. 이 위험은, 다중 프로세스가 이 파일류 객체에 *close()*를 호출하면, 같은 데이터가 객체에 여러 번 플러시 되도록 만들어 손상을 일으킬 수 있다는 것입니다.

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

fork 시작 방법에는 적용되지 않는 몇 가지 추가 제한 사항이 있습니다.

더 높은 피클 가능성

> `Process.__init__()` 에 대한 모든 인자가 피클 가능한지 확인하십시오. 또한, *Process* 의 서브 클래스를 만들면, *Process.start* 메서드가 호출될 때 그 인스턴스가 피클 가능하도록 해야 합니다.

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

> 메인 모듈이 의도하지 않은 부작용(가령 새 프로세스 시작)을 일으키지 않고 새 파이썬 인터프리터가 안전하게 임포트 할 수 있는지 확인하십시오.
>
> 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.

17.2.4 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()
    manager.start()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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
        )

def calculatestar(args):
    return calculate(*args)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                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):
    time.sleep(0.5*random.random())
    return a * b
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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()
```

17.3 concurrent 패키지

현재, 이 패키지에는 하나의 모듈만 있습니다.:

- *concurrent.futures* – 병렬 작업 시작하기

17.4 concurrent.futures — 병렬 작업 실행하기

버전 3.2에 추가.

소스 코드: Lib/concurrent/futures/thread.py와 Lib/concurrent/futures/process.py

---

concurrent.futures 모듈은 비동기적으로 콜러블을 실행하는 고수준 인터페이스를 제공합니다.

비동기 실행은 (*ThreadPoolExecutor*를 사용해서) 스레드나 (*ProcessPoolExecutor*를 사용해서) 별도의 프로세스로 수행 할 수 있습니다. 둘 다 추상 *Executor* 클래스로 정의된 것과 같은 인터페이스를 구현합니다.

17.4.1 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(*func*, **iterables*, *timeout=None*, *chunksize=1*)

*map(func, *iterables)* 과 비슷하지만, 다음과 같은 차이가 있습니다:

- *iterables* 는 느긋하게 처리되는 것이 아니라 즉시 수집됩니다.
- *func* 는 비동기적으로 실행되며 *func* 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 *concurrent.futures.TimeoutError* 를 발생시킵니다. *timeout* 은 int 또는 float가 될 수 있습니다. *timeout* 이 지정되지 않았거나 `None` 인 경우, 대기 시간에는 제한이 없습니다.

func 호출이 예외를 일으키면, 값을 이터레이터에서 꺼낼 때 해당 예외가 발생합니다.

*ProcessPoolExecutor*를 사용할 때, 이 메서드는 *iterables* 를 다수의 덩어리로 잘라서 별도의 작업으로 풀에 제출합니다. 이러한 덩어리의 (대략적인) 크기는 *chunksize* 를 양의 정수로 설정하여 지정할 수 있습니다. 매우 긴 이터러블의 경우 *chunksize* 에 큰 값을 사용하면 기본 크기인 1에 비해 성능이 크게 향상될 수 있습니다. *ThreadPoolExecutor* 의 경우, *chunksize* 는 아무런 효과가 없습니다.

버전 3.5에서 변경: *chunksize* 인자가 추가되었습니다.

shutdown(*wait=True*)

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.

`with` 문을 사용하여 *Executor*를 종료시키면 (*Executor.shutdown()* 를 *wait* 값 `True` 로 호출한 것처럼 대기합니다), 이 메서드를 명시적으로 호출할 필요가 없어집니다.:

```
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')
```

17.4.2 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 will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    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)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    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.

initializer 는 각 작업자 스레드의 시작 부분에서 호출되는 선택적 콜러블입니다; *initargs* 는 initializer에 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 3.6에 추가: *thread_name_prefix* 인자가 추가되어, 디버깅 편의를 위해 사용자가 풀이 만드는 작업자 스레드의 *threading.Thread* 이름을 제어 할 수 있습니다.

버전 3.7에서 변경: *initializer* 및 *initargs* 인자가 추가되었습니다.

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://nonexistant-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    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)))
```

17.4.3 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.

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*)

최대 *max_workers* 프로세스의 풀을 사용하여 호출을 비동기적으로 실행하는 *Executor* 서브 클래스. *max_workers* 가 None 이거나 주어지지 않았다면, 기계의 프로세서 수를 기본값으로 사용합니다. *max_workers* 가 0 보다 작거나 같으면 *ValueError* 가 발생합니다. 윈도우에서, *max_workers*는 61보다 작거나 같아야 합니다. 그렇지 않으면 *ValueError*가 발생합니다. *max_workers*가 None 이면, 더 많은 프로세서를 사용할 수 있다 할지라도 선택된 기본값은 최대 61이 될 것입니다. *mp_context* 는 multiprocessing 컨텍스트이거나 None일 수 있습니다. 작업자들을 만드는데 사용될 것입니다. *mp_context* 가 None 이거나 주어지지 않으면 기본 multiprocessing 컨텍스트가 사용됩니다.

initializer 는 각 작업자 프로세스의 시작 부분에서 호출되는 선택적 콜러블입니다; *initargs* 는 initializer에 전달되는 인자들의 튜플입니다. *initializer* 가 예외를 발생시키는 경우, 현재 계류 중인 모든 작업과 풀에 추가로 작업을 제출하려는 시도는 *BrokenProcessPool* 을 발생시킵니다.

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.

ProcessPoolExecutor 예제

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    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()
```

17.4.4 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 *concurrent.futures.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 *concurrent.futures.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.

set_exception(*exception*)
*Future*와 관련된 작업 결과를 *Exception* *exception* 으로 설정합니다.

이 메서드는 *Executor* 구현과 단위 테스트에서만 사용해야 합니다.

17.4.5 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. 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
FIRST_COMPLETED	퓨처가 어느 하나라도 끝나거나 취소될 때 함수가 반환됩니다.
FIRST_EXCEPTION	어느 한 퓨처가 예외를 일으켜 완료하면 함수가 반환됩니다. 어떤 퓨처도 예외를 발생시키지 않으면 ALL_COMPLETED와 같습니다.
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 일드됩니다. 반환된 이터레이터는, *__next__()* 가 호출되고, *as_completed()* 호출 시점으로부터 *timeout* 초 후에 결과를 얻을 수 없는 경우 *concurrent.futures.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.

17.4.6 예외 클래스

exception concurrent.futures.**CancelledError**
퓨처가 취소될 때 발생합니다.

exception concurrent.futures.**TimeoutError**
퓨처 연산이 지정된 제한시간을 초과할 때 발생합니다.

exception concurrent.futures.**BrokenExecutor**
RuntimeError 에서 파생됩니다, 이 예외 클래스는 어떤 이유로 실행기가 망가져서 새 작업을 제출하거나 실행할 수 없을 때 발생합니다.

버전 3.7에 추가.

exception concurrent.futures.thread.**BrokenThreadPool**
BrokenExecutor 에서 파생됩니다, 이 예외 클래스는 ThreadPoolExecutor 의 작업자 중 하나가 초기화에 실패했을 때 발생합니다.

버전 3.7에 추가.

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.

버전 3.3에 추가.

17.5 subprocess — Subprocess management

Source code: Lib/subprocess.py

The *subprocess* module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

Information about how the *subprocess* module can be used to replace these modules and functions can be found in the following sections.

더 보기:

PEP 324 – PEP proposing the subprocess module

17.5.1 Using the subprocess Module

The recommended approach to invoking subprocesses is to use the *run()* function for all use cases it can handle. For more advanced use cases, the underlying *Popen* interface can be used directly.

The *run()* function was added in Python 3.5; if you need to retain compatibility with older versions, see the *Older high-level API* section.

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*)
Run the command described by *args*. Wait for command to complete, then return a *CompletedProcess* instance.

The arguments shown above are merely the most common ones, described below in *Frequently Used Arguments* (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the `Popen` constructor - most of the arguments to this function are passed through to that interface. (*timeout*, *input*, *check*, and *capture_output* are not.)

If *capture_output* is true, stdout and stderr will be captured. When used, the internal `Popen` object is automatically created with `stdout=PIPE` and `stderr=PIPE`. The *stdout* and *stderr* arguments may not be supplied at the same time as *capture_output*. If you wish to capture and combine both streams into one, use `stdout=PIPE` and `stderr=STDOUT` instead of *capture_output*.

The *timeout* argument is passed to `Popen.communicate()`. If the timeout expires, the child process will be killed and waited for. The `TimeoutExpired` exception will be re-raised after the child process has terminated.

The *input* argument is passed to `Popen.communicate()` and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *text* is true. When used, the internal `Popen` object is automatically created with `stdin=PIPE`, and the *stdin* argument may not be used as well.

If *check* is true, and the process exits with a non-zero exit code, a `CalledProcessError` exception will be raised. Attributes of that exception hold the arguments, the exit code, and stdout and stderr if they were captured.

If *encoding* or *errors* are specified, or *text* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the `io.TextIOWrapper` default. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to `Popen`.

Examples:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
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'')
```

버전 3.5에 추가.

버전 3.6에서 변경: Added *encoding* and *errors* parameters

버전 3.7에서 변경: Added the *text* parameter, as a more understandable alias of *universal_newlines*. Added the *capture_output* parameter.

버전 3.7.17에서 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**

The return value from `run()`, representing a process that has finished.

args

The arguments used to launch the process. This may be a list or a string.

returncode

Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

stdout
Captured stdout from the child process. A bytes sequence, or a string if *run()* was called with an encoding, errors, or text=True. None if stdout was not captured.

If you ran the process with stderr=subprocess.STDOUT, stdout and stderr will be combined in this attribute, and *stderr* will be None.

stderr
Captured stderr from the child process. A bytes sequence, or a string if *run()* was called with an encoding, errors, or text=True. None if stderr was not captured.

check_returncode()
If *returncode* is non-zero, raise a *CalledProcessError*.

버전 3.5에 추가.

subprocess.**DEVNULL**
Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *Popen* and indicates that the special file *os.devnull* will be used.

버전 3.3에 추가.

subprocess.**PIPE**
Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *Popen* and indicates that a pipe to the standard stream should be opened. Most useful with *Popen.communicate()*.

subprocess.**STDOUT**
Special value that can be used as the *stderr* argument to *Popen* and indicates that standard error should go into the same handle as standard output.

exception subprocess.**SubprocessError**
Base class for all other exceptions from this module.

버전 3.3에 추가.

exception subprocess.**TimeoutExpired**
Subclass of *SubprocessError*, raised when a timeout expires while waiting for a child process.

cmd
Command that was used to spawn the child process.

timeout
Timeout in seconds.

output
Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, None.

stdout
Alias for output, for symmetry with *stderr*.

stderr
Stderr output of the child process if it was captured by *run()*. Otherwise, None.

버전 3.3에 추가.

버전 3.5에서 변경: *stdout* and *stderr* attributes added

exception subprocess.**CalledProcessError**
Subclass of *SubprocessError*, raised when a process run by *check_call()* or *check_output()* returns a non-zero exit status.

returncode
Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

cmd
Command that was used to spawn the child process.

output
Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, None.

stdout
Alias for output, for symmetry with *stderr*.

stderr
Stderr output of the child process if it was captured by *run()*. Otherwise, None.

버전 3.5에서 변경: *stdout* and *stderr* attributes added

Frequently Used Arguments

To support a wide variety of use cases, the *Popen* constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

args is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either *shell* must be *True* (see below) or else the string must simply name the program to be executed without specifying any arguments.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing file object, and None. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

If *encoding* or *errors* are specified, or *text* (also known as *universal_newlines*) is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for *io.TextIOWrapper*.

For *stdin*, line ending characters '\n' in the input will be converted to the default line separator *os.linesep*. For *stdout* and *stderr*, all line endings in the output will be converted to '\n'. For more information see the documentation of the *io.TextIOWrapper* class when the *newline* argument to its constructor is None.

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

버전 3.6에 추가: Added *encoding* and *errors* parameters.

버전 3.7에 추가: Added the *text* parameter as an alias for *universal_newlines*.

참고: The newlines attribute of the file objects *Popen.stdin*, *Popen.stdout* and *Popen.stderr* are not updated by the *Popen.communicate()* method.

If *shell* is True, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, *glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()*, and *shutil*).

버전 3.3에서 변경: When *universal_newlines* is `True`, the class uses the encoding *locale.getpreferredencoding(False)* instead of `locale.getpreferredencoding()`. See the *io.TextIOWrapper* class for more information on this change.

참고: Read the *Security Considerations* section before using `shell=True`.

These options, along with all of the other options, are described in more detail in the *Popen* constructor documentation.

Popen Constructor

The underlying process creation and management in this module is handled by the *Popen* class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

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=()*, ***, *encoding=None*, *errors=None*, *text=None*)

Execute a child program in a new process. On POSIX, the class uses *os.execvp()*-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to *Popen* are as follows.

args should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

An example of passing some arguments to an external program as a sequence is:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

On POSIX, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

참고: It may not be obvious how to break a shell command into a sequence of arguments, especially in complex cases. *shlex.split()* can illustrate how to determine the correct tokenization for *args*:

```
>>> 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) # Success!
```

Note in particular that options (such as *-input*) and arguments (such as *eggs.txt*) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the *echo* command shown above) are single list elements.

On Windows, if *args* is a sequence, it will be converted to a string in a manner described in *Converting an argument sequence to a string on Windows*. This is because the underlying `CreateProcess()` operates on strings.

The *shell* argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If *shell* is `True`, it is recommended to pass *args* as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, *Popen* does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the `COMSPEC` environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. **`dir`** or **`copy`**). You do not need `shell=True` to run a batch file or console-based executable.

참고: Read the *Security Considerations* section before using `shell=True`.

bufsize will be supplied as the corresponding argument to the *open()* function when creating the stdin/stdout/stderr pipe file objects:

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `universal_newlines=True` i.e., in a text mode)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used.

버전 3.3.1에서 변경: *bufsize* now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as **`ps`**. If `shell=True`, on POSIX the *executable* argument specifies a replacement shell for the default `/bin/sh`.

버전 3.6에서 변경: *executable* parameter accepts a *path-like object* on POSIX.

버전 3.7.17에서 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* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing *file object*, and `None`. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

경고: The *preexec_fn* parameter is not safe to use in the presence of threads in your application. The child process could deadlock before exec is called. If you must use it, keep it trivial! Minimize the number of libraries you call into.

참고: If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* parameter can take the place of a previously common use of *preexec_fn* to call os.setsid() in the child.

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in 파일 기술자의 상속.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of *`STARTUPINFO.lpAttributeList`*, or by standard handle redirection.

버전 3.2에서 변경: The default for *close_fds* was changed from *`False`* to what is described above.

버전 3.7에서 변경: On Windows the default for *close_fds* was changed from *`False`* to *`True`* when redirecting the standard handles. It's now possible to set *close_fds* to *`True`* when redirecting the standard handles.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be *`True`*. (POSIX only)

버전 3.2에 추가: The *pass_fds* parameter was added.

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a *`str`* and *path-like* object. In particular, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

버전 3.6에서 변경: *cwd* parameter accepts a *path-like object*.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

버전 3.2에서 변경: *restore_signals* was added.

If *start_new_session* is true the setsid() system call will be made in the child process prior to the execution of the subprocess. (POSIX only)

버전 3.2에서 변경: *start_new_session* was added.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment.

참고: If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified encoding and *errors*, as described above in *Frequently Used Arguments*. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

버전 3.6에 추가: *encoding* and *errors* were added.

버전 3.7에 추가: *text* was added as a more readable alias for *universal_newlines*.

If given, *startupinfo* will be a *`STARTUPINFO`* object, which is passed to the underlying `CreateProcess` function. *creationflags*, if given, can be one or more of the following flags:

- *`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*

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

버전 3.2에서 변경: Added context manager support.

버전 3.6에서 변경: Popen destructor now emits a *ResourceWarning* warning if the child process is still running.

Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.

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.

A *ValueError* will be raised if *Popen* is called with invalid arguments.

check_call() and *check_output()* will raise *CalledProcessError* if the called process returns a non-zero return code.

All of the functions and methods that accept a *timeout* parameter, such as *call()* and *Popen.communicate()* will raise *TimeoutExpired* if the timeout expires before the process exits.

Exceptions defined in this module all inherit from *SubprocessError*.

버전 3.3에 추가: The *SubprocessError* base class was added.

17.5.2 Security Considerations

Unlike some other popen functions, this implementation will never implicitly call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities.

When using `shell=True`, the *shlex.quote()* function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

17.5.3 Popen Objects

Instances of the *Popen* class have the following methods:

Popen.**poll**()
Check if child process has terminated. Set and return *returncode* attribute. Otherwise, returns None.

Popen.**wait**(*timeout=None*)
Wait for child process to terminate. Set and return *returncode* attribute.

If the process does not terminate after *timeout* seconds, raise a *TimeoutExpired* exception. It is safe to catch this exception and retry the wait.

참고: This will deadlock when using stdout=PIPE or stderr=PIPE and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use *Popen.communicate()* when using pipes to avoid that.

참고: The function is implemented using a busy loop (non-blocking call and short sleeps). Use the *asyncio* module for an asynchronous wait: see *asyncio.create_subprocess_exec*.

버전 3.3에서 변경: *timeout* was added.

Popen.**communicate**(*input=None*, *timeout=None*)
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be data to be sent to the child process, or None, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

communicate() returns a tuple (stdout_data, stderr_data). The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process's stdin, you need to create the Popen object with stdin=PIPE. Similarly, to get anything other than None in the result tuple, you need to give stdout=PIPE and/or stderr=PIPE too.

If the process does not terminate after *timeout* seconds, a *TimeoutExpired* exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

참고: The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

버전 3.3에서 변경: *timeout* was added.

Popen.**send_signal**(*signal*)
Sends the signal *signal* to the child.

참고: On Windows, SIGTERM is an alias for *terminate()*. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes *CREATE_NEW_PROCESS_GROUP*.

Popen.**terminate**()
Stop the child. On POSIX OSs the method sends SIGTERM to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

Popen.**kill**()
Kills the child. On POSIX OSs the function sends SIGKILL to the child. On Windows *kill()* is an alias for *terminate()*.

The following attributes are also available:

Popen.**args**
The *args* argument as it was passed to *Popen* – a sequence of program arguments or else a single string.

버전 3.3에 추가.

Popen.**stdin**
If the *stdin* argument was *PIPE*, this attribute is a writeable stream object as returned by *open()*. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not *PIPE*, this attribute is `None`.

Popen.**stdout**
If the *stdout* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not *PIPE*, this attribute is `None`.

Popen.**stderr**
If the *stderr* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not *PIPE*, this attribute is `None`.

경고: Use *communicate()* rather than *.stdin.write*, *.stdout.read* or *.stderr.read* to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen.**pid**
The process ID of the child process.

Note that if you set the *shell* argument to `True`, this is the process ID of the spawned shell.

Popen.**returncode**
The child return code, set by *poll()* and *wait()* (and indirectly by *communicate()*). A `None` value indicates that the process hasn't terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

17.5.4 Windows Popen Helpers

The *STARTUPINFO* class and following constants are only available on Windows.

class subprocess.**STARTUPINFO**(*, *dwFlags=0*, *hStdInput=None*, *hStdOutput=None*, *hStdError=None*, *wShowWindow=0*, *lpAttributeList=None*)

Partial support of the Windows STARTUPINFO structure is used for *Popen* creation. The following attributes can be set by passing them as keyword-only arguments.

버전 3.7에서 변경: Keyword-only argument support was added.

dwFlags

A bit field that determines whether certain *STARTUPINFO* attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard input handle for the process. If *STARTF_USESTDHANDLES* is not specified, the default for standard input is the keyboard buffer.

hStdOutput

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.

hStdError

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

wShowWindow

If *dwFlags* specifies *STARTF_USESHOWWINDOW*, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the ShowWindow function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

SW_HIDE is provided for this attribute. It is used when *Popen* is called with `shell=True`.

lpAttributeList

A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see UpdateProcThreadAttribute.

Supported attributes:

handle_list Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by *os.set_handle_inheritable()* when passed to the *Popen* constructor, else *OSError* will be raised with Windows error `ERROR_INVALID_PARAMETER` (87).

> **경고:** In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as *os.system()*. This also applies to standard handle redirection, which temporarily creates inheritable handles.

버전 3.7에 추가.

Windows Constants

The *subprocess* module exposes the following constants.

subprocess.**STD_INPUT_HANDLE**
The standard input device. Initially, this is the console input buffer, CONIN$.

subprocess.**STD_OUTPUT_HANDLE**
The standard output device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.**STD_ERROR_HANDLE**
The standard error device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.**SW_HIDE**
Hides the window. Another window will be activated.

subprocess.**STARTF_USESTDHANDLES**
Specifies that the *STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput*, and *STARTUPINFO.hStdError* attributes contain additional information.

subprocess.**STARTF_USESHOWWINDOW**
Specifies that the *STARTUPINFO.wShowWindow* attribute contains additional information.

subprocess.**CREATE_NEW_CONSOLE**
The new process has a new console, instead of inheriting its parent's console (the default).

subprocess.**CREATE_NEW_PROCESS_GROUP**
A *Popen* `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using *os.kill()* on the subprocess.

This flag is ignored if *CREATE_NEW_CONSOLE* is specified.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have an above average priority.

버전 3.7에 추가.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have a below average priority.

버전 3.7에 추가.

subprocess.**HIGH_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have a high priority.

버전 3.7에 추가.

subprocess.**IDLE_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have an idle (lowest) priority.

버전 3.7에 추가.

subprocess.**NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have an normal priority. (default)

버전 3.7에 추가.

subprocess.**REALTIME_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have realtime priority. You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

버전 3.7에 추가.

subprocess.**CREATE_NO_WINDOW**
A *Popen* creationflags parameter to specify that a new process will not create a window.

버전 3.7에 추가.

subprocess.**DETACHED_PROCESS**
A *Popen* creationflags parameter to specify that a new process will not inherit its parent's console. This value cannot be used with CREATE_NEW_CONSOLE.

버전 3.7에 추가.

subprocess.**CREATE_DEFAULT_ERROR_MODE**
A *Popen* creationflags parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

버전 3.7에 추가.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**
A *Popen* creationflags parameter to specify that a new process is not associated with the job.

버전 3.7에 추가.

17.5.5 Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to subprocess. You can now use *run()* in many cases, but lots of existing code calls these functions.

subprocess.**call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)
Run the command described by *args*. Wait for command to complete, then return the *returncode* attribute.

Code needing to capture stdout or stderr should use *run()* instead:

run(…).returncode

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

참고: Do not use stdout=PIPE or stderr=PIPE with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

버전 3.3에서 변경: *timeout* was added.

버전 3.7.17에서 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*)
Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute.

Code needing to capture stdout or stderr should use *run()* instead:

run(…, check=True)

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

참고: Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

버전 3.3에서 변경: *timeout* was added.

버전 3.7.17에서 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*)

Run command with arguments and return its output.

If the return code was non-zero it raises a *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute and any output in the *output* attribute.

This is equivalent to:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of *run()* - most arguments are passed directly through to that interface. However, explicitly passing `input=None` to inherit the parent's standard input file handle is not supported.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *text*, *encoding*, *errors*, or *universal_newlines* to `True` as described in *Frequently Used Arguments* and *run()*.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

버전 3.1에 추가.

버전 3.3에서 변경: *timeout* was added.

버전 3.4에서 변경: Support for the *input* keyword argument was added.

버전 3.6에서 변경: *encoding* and *errors* were added. See *run()* for details.

버전 3.7에 추가: *text* was added as a more readable alias for *universal_newlines*.

버전 3.7.17에서 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.

17.5.6 Replacing Older Functions with the subprocess Module

In this section, "a becomes b" means that b can be used as a replacement for a.

참고: All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise *OSError* instead.

In addition, the replacements using *check_output()* will fail with a *CalledProcessError* if the requested operation produces a non-zero return code. The output is still available as the *output* attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the *subprocess* module.

Replacing /bin/sh shell backquote

```
output=`mycmd myarg`
```

becomes:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=`dmesg | grep hda`
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=`dmesg | grep hda`
```

becomes:

```
output=check_output("dmesg | grep hda", shell=True)
```

Replacing os.system()

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```
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)
```

Replacing the `os.spawn` family

P_NOWAIT example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
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")
```

Replacing functions from the `popen2` module

참고: If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` and `popen2.Popen4` basically work as *`subprocess.Popen`*, except that:

- *`Popen`* raises an exception if the execution fails.
- The *capturestderr* argument is replaced with the *stderr* argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with *`Popen`* to guarantee this behavior on all platforms or past Python versions.

17.5.7 Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

`subprocess.`**`getstatusoutput`**(*cmd*)
Return `(exitcode, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(exitcode, output)`. The locale encoding is used; see the notes on *Frequently Used Arguments* for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

```
>>> 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, '')
```

Availability: POSIX & Windows.

버전 3.3.4에서 변경: Windows support was added.

The function now returns (exitcode, output) instead of (status, output) as it did in Python 3.3.3 and earlier. exitcode has the same value as *`returncode`*.

`subprocess.`**`getoutput`**(*cmd*)
Return output (stdout and stderr) of executing *cmd* in a shell.

Like *`getstatusoutput()`*, except the exit code is ignored and the return value is a string containing the command's output. Example:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: POSIX & Windows.

버전 3.3.4에서 변경: Windows support added

17.5.8 Notes

Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.

5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

더 보기:

shlex Module which provides function to parse and escape command lines.

17.6 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',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.6.1 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 숫자 형이어야 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 3.3에 추가: *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 3.3에 추가: *kwargs* 매개 변수가 추가되었습니다.

scheduler.**cancel**(*event*)
큐에서 이벤트를 제거합니다. *event*가 현재 큐에 없으면, 이 메서드는 *ValueError*를 발생시킵니다.

scheduler.**empty**()
Return `True` if the event queue is empty.

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 3.3에 추가: *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.

17.7 queue — 동기화된 큐 클래스

소스 코드: Lib/queue.py

The *queue* module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The *Queue* class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the *threading* module.

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 *SimpleQueue*를 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 `sorted(list(entries))[0]` 에 의해 반환되는 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.

버전 3.7에 추가.

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.

17.7.1 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 `True`를, 그렇지 않으면 `False`를 반환합니다. empty()가 `True`를 반환하면, put()에 대한 후속 호출이 블록 되지 않는다고 보장하는 것은 아닙니다. 마찬가지로 empty()가 `False`를 반환하면, get()에 대한 후속 호출이 블록 되지 않는다고 보장하는 것은 아닙니다.

Queue.**full**()
큐가 가득 차면 `True`를, 그렇지 않으면 `False`를 반환합니다. full()이 `True`를 반환하면, get()에 대한 후속 호출이 블록 되지 않는다고 보장하는 것은 아닙니다. 마찬가지로 full()이 `False`를 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).

Queue.**put_nowait**(*item*)
`put(item, 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 `None`이면, 이 연산은 하부 록에 대한 중단되지 않는(uninterruptible) 대기로 들어갑니다. 이는 어떤 예외도 발생할 수 없음을 뜻하고, 특히 SIGINT가 *KeyboardInterrupt*를 일으키지 않습니다.

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.

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:

```
def worker():
    while True:
        item = q.get()
        if item is None:
            break
        do_work(item)
        q.task_done()

q = queue.Queue()
threads = []
for i in range(num_worker_threads):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)

for item in source():
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(None)
for t in threads:
    t.join()
```

17.7.2 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 implementation detail: 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)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 is an alternative implementation of unbounded queues with fast atomic *append()* and *popleft()* operations that do not require locking.

다음은 위 서비스 중 일부에 대한 지원 모듈입니다:

17.8 _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, 스택 트레이스가 인쇄된 다음 스레드가 종료합니다 (하지만 다른 스레드는 계속 실행됩니다).

_thread.**interrupt_main**()
메인 스레드에 도달한 *signal.SIGINT* 시그널의 효과를 시뮬레이트합니다. 스레드는 이 함수를 사용하여 메인 스레드를 인터럽트 할 수 있습니다.

*signal.SIGINT*가 파이썬에 의해 처리되지 않으면 (*signal.SIG_DFL*이나 *signal.SIG_IGN*으로 설정됩니다), 이 함수는 아무것도 하지 않습니다.

_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.**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, *ValueError*가 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: 윈도우, POSIX 스레드가 있는 시스템.

_thread.**TIMEOUT_MAX**
`Lock.acquire()`의 *timeout* 매개 변수에 허용되는 최댓값. 이 값보다 큰 timeout을 지정하면 *OverflowError*가 발생합니다.

버전 3.2에 추가.

록 객체에는 다음과 같은 메서드가 있습니다:

lock.**acquire**(*waitflag=1*, *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).

정수 *waitflag* 인자가 있으면, 행동은 그 값에 따라 다릅니다: 0이면 대기하지 않고 즉시 획득할 수 있을 때만 록이 획득되고, 0이 아니면 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. *waitflag*이 0이면 *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.

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:

- 스레드는 이상한 방식으로 인터럽트와 상호 작용합니다: *KeyboardInterrupt* 예외는 임의의 스레드가 수신합니다. (*signal* 모듈을 사용할 수 있으면, 인터럽트는 항상 메인 스레드로 갑니다.)
- *sys.exit()*를 호출하거나 *SystemExit* 예외를 발생시키는 것은 *_thread.exit()*를 호출하는 것과 동등합니다.
- 록에 대한 `acquire()` 메서드를 인터럽트 할 수 없습니다 — 록이 획득된 후에 *KeyboardInterrupt* 예외가 발생합니다.
- 메인 스레드가 종료할 때, 다른 스레드가 살아남는지는 시스템이 정의합니다. 대부분 시스템에서, `try … finally` 절을 실행하거나 객체 파괴자(destructor)를 실행하지 않고 강제 종료됩니다.
- 메인 스레드가 종료할 때, (`try … finally` 절이 적용되는 것을 제외하고는) 일반적인 정리 작업을 수행하지 않으며, 표준 I/O 파일은 플러시 되지 않습니다.

17.9 _dummy_thread — Drop-in replacement for the _thread module

Source code: Lib/_dummy_thread.py

버전 3.7부터 폐지: Python now always has threading enabled. Please use *_thread* (or, better, *threading*) instead.

This module provides a duplicate interface to the *_thread* module. It was meant to be imported when the *_thread* module was not provided on a platform.

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

17.10 dummy_threading — Drop-in replacement for the threading module

Source code: Lib/dummy_threading.py

버전 3.7부터 폐지: Python now always has threading enabled. Please use *threading* instead.

This module provides a duplicate interface to the *threading* module. It was meant to be imported when the *_thread* module was not provided on a platform.

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

CHAPTER 18

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.

버전 3.7에 추가.

18.1 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.

버전 3.7.1에 추가.

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')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.**Token**
Token 객체는 *ContextVar.set()* 메서드에 의해 반환됩니다. *ContextVar.reset()* 메서드에 전달해서 변수의 값을 해당 *set* 이전의 값으로 되돌릴 수 있습니다.

Token.**var**
읽기 전용 프로퍼티. 토큰을 생성 한 *ContextVar* 객체를 가리 킵니다.

Token.**old_value**
읽기 전용 프로퍼티. 토큰을 생성 한 *ContextVar.set()* 메서드 호출 전 변수의 값으로 설정됩니다. Token.MISSING 은 호출 전에 변수가 설정되지 않았음을 나타냅니다.

Token.**MISSING**
Token.old_value 에 의해 사용되는 표지 객체.

18.2 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: Context = 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.

Context는 *collections.abc.Mapping* 인터페이스를 구현합니다.

run(*callable*, **args*, ***kwargs*)

run 메서드가 호출된 컨텍스트 객체에서 callable(*args, **kwargs) 코드를 실행합니다. 실행 결과를 반환하거나 예외가 발생하면 예외를 전파합니다.

callable 이 만드는 모든 컨텍스트 변수에 대한 변경 사항은 컨텍스트 개체에 포함됩니다:

```
var = ContextVar('var')
var.set('spam')

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
# var.get() == 'spam'
```

이 메서드는 둘 이상의 OS 스레드에서 같은 컨텍스트 객체에 대해 호출될 때나 재귀적으로 호출될 때 *RuntimeError* 를 발생시킵니다.

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.

18.3 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():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
        writer.write(line)

    writer.write(render_goodbye())
    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())
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# To test it you can use telnet:
#     telnet 127.0.0.1 8081
```

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!')

# Python 3.7+
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.

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.

레퍼런스

19.1.1 코루틴과 태스크

이 절에서는 코루틴과 태스크로 작업하기 위한 고급 asyncio API에 관해 설명합니다.

코루틴

async/await 문법으로 선언된 코루틴은 asyncio 응용 프로그램을 작성하는 기본 방법입니다. 예를 들어, 다음 코드 조각(파이썬 3.7 이상 필요)은 "hello"를 인쇄하고, 1초 동안 기다린 다음, "world"를 인쇄합니다:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     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 세 가지 주요 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)

  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')}")

      # Wait until both tasks are completed (should take
      # around 2 seconds.)
  ```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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
```

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():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

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.

---

asyncio는 기존 제너레이터 기반 코루틴도 지원합니다.

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():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    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()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

Future 객체를 반환하는 저수준 함수의 좋은 예는 *loop.run_in_executor()*입니다.

asyncio 프로그램 실행하기

asyncio.**run**(*coro*, *, *debug=False*)

Execute the *coroutine* *coro* and return the result.

이 함수는 전달된 코루틴을 실행하고, asyncio 이벤트 루프와 비동기 제너레이터의 파이널리제이션을 관리합니다.

다른 asyncio 이벤트 루프가 같은 스레드에서 실행 중일 때, 이 함수를 호출할 수 없습니다.

*debug*이 `True`면, 이벤트 루프가 디버그 모드로 실행됩니다.

이 함수는 항상 새 이벤트 루프를 만들고 끝에 이벤트 루프를 닫습니다. asyncio 프로그램의 메인 진입 지점으로 사용해야 하고, 이상적으로는 한 번만 호출해야 합니다.

예:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

버전 3.7에 추가: **중요:** 이 함수는 파이썬 3.7에서 잠정적으로 asyncio에 추가되었습니다.

태스크 만들기

asyncio.**create_task**(*coro*)

coro 코루틴을 *Task*로 감싸고 실행을 예약합니다. Task 객체를 반환합니다.

*get_running_loop()*에 의해 반환된 루프에서 태스크가 실행되고, 현재 스레드에 실행 중인 루프가 없으면 *RuntimeError*가 발생합니다.

이 함수는 **파이썬 3.7에서 추가되었습니다.** 파이썬 3.7 이전 버전에서는, 대신 저수준 *asyncio.ensure_future()* 함수를 사용할 수 있습니다:

```
async def coro():
    ...

# In Python 3.7+
task = asyncio.create_task(coro())
...

# This works in all Python versions but is less readable
task = asyncio.ensure_future(coro())
...
```

버전 3.7에 추가.

잠자기

coroutine asyncio.**sleep**(*delay*, *result=None*, *, *loop=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.

loop 인자는 폐지되었고 파이썬 3.10에서 삭제 예정입니다.

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())
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

동시에 태스크 실행하기

awaitable asyncio.**gather**(**aws*, *loop=None*, *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.

예:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2)...
#     Task B: Compute factorial(2)...
#     Task C: Compute factorial(2)...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3)...
#     Task C: Compute factorial(3)...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
#      Task B: factorial(3) = 6
#      Task C: Compute factorial(4)...
#      Task C: factorial(4) = 24
```

버전 3.7에서 변경: *gather* 자체가 취소되면, *return_exceptions*와 관계없이 취소가 전파됩니다.

취소로부터 보호하기

awaitable asyncio.**shield**(*aw*, *, *loop=None*)

어웨이터블 객체를 취소로부터 보호합니다.

*aw*가 코루틴이면 자동으로 태스크로 예약됩니다.

다음 문장:

```
res = await shield(something())
```

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:

```
try:
    res = await shield(something())
except CancelledError:
    res = None
```

시간제한

coroutine asyncio.**wait_for**(*aw*, *timeout*, *, *loop=None*)

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 *asyncio.TimeoutError*를 발생시킵니다.

태스크 취소를 피하려면, *shield()*로 감싸십시오.

이 함수는 퓨처가 실제로 취소될 때까지 대기하므로, 총 대기 시간이 *timeout*을 초과할 수 있습니다.

대기가 취소되면, 퓨처 *aw*도 취소됩니다.

loop 인자는 폐지되었고 파이썬 3.10에서 삭제 예정입니다.

예:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

버전 3.7에서 변경: 시간 초과로 인해 *aw*가 취소되면, `wait_for`는 *aw*가 취소될 때까지 대기합니다. 이전에는 *asyncio.TimeoutError*가 즉시 발생했습니다.

대기 프리미티브

coroutine asyncio.**wait**(*aws*, ***, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

aws 집합에 있는 어웨이터블 객체를 동시에 실행하고, *return_when*에 의해 지정된 조건을 만족할 때까지 블록합니다.

*aws*에 있는 어웨이터블이 코루틴이면, 자동으로 태스크로 예약됩니다. 코루틴 객체를 `wait()`로 직접 전달하는 것은 혼란스러운 동작으로 연결되므로 폐지되었습니다.

두 집합의 태스크/퓨처를 반환합니다: `(done, pending)`.

사용법:

```
done, pending = await asyncio.wait(aws)
```

loop 인자는 폐지되었고 파이썬 3.10에서 삭제 예정입니다.

timeout(float나 int)을 지정하면, 반환하기 전에 대기할 최대 시간(초)을 제어할 수 있습니다.

이 함수는 *asyncio.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
`FIRST_COMPLETED`	퓨처가 하나라도 끝나거나 취소될 때 함수가 반환됩니다.
`FIRST_EXCEPTION`	퓨처가 하나라도 예외를 일으켜 끝나면 함수가 반환됩니다. 어떤 퓨처도 예외를 일으키지 않으면 `ALL_COMPLETED`와 같습니다.
`ALL_COMPLETED`	모든 퓨처가 끝나거나 취소되면 함수가 반환됩니다.

*wait_for()*와 달리, `wait()`는 시간 초과가 발생할 때 퓨처를 취소하지 않습니다.

참고: `wait()`는 코루틴을 태스크로 자동 예약하고, 나중에 묵시적으로 생성된 Task 객체를 `(done, pending)` 집합으로 반환합니다. 따라서 다음 코드는 기대한 대로 작동하지 않습니다:

```
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

위의 조각을 고치는 방법은 다음과 같습니다:

```
async def foo():
    return 42

task = asyncio.create_task(foo())
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```

코루틴 객체를 wait() 로 직접 전달하는 것은 폐지되었습니다.

---

asyncio.**as_completed**(*aws*, *, *loop=None*, *timeout=None*)

aws 집합에 있는 어웨이터블 객체를 동시에 실행합니다. *Future* 객체의 이터레이터를 반환합니다. 반환된 각 Future 객체는 남아있는 어웨이터블 집합의 가장 빠른 결과를 나타냅니다.

모든 퓨처가 완료되기 전에 시간 초과가 발생하면 *asyncio.TimeoutError*를 발생시킵니다.

예:

```
for f in as_completed(aws):
    earliest_result = await f
    # ...
```

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:

```
# 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) == 3
```

코루틴에서 예외가 발생하면, 반환된 Future에 통지됩니다. 또한, 이벤트 루프에서 태스크를 취소하는 데 사용할 수 있습니다:

```
try:
    result = future.result(timeout)
except asyncio.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.

버전 3.5.1에 추가.

인트로스팩션

asyncio.**current_task**(*loop=None*)
현재 실행 중인 *Task* 인스턴스를 반환하거나 태스크가 실행되고 있지 않으면 None을 반환합니다.

*loop*가 None이면, 현재 루프를 가져오는 데 *get_running_loop()*가 사용됩니다.

버전 3.7에 추가.

asyncio.**all_tasks**(*loop=None*)
루프에 의해 실행되는 아직 완료되지 않은 *Task* 객체 집합을 반환합니다.

*loop*가 None이면, 현재 루프를 가져오는 데 *get_running_loop()*가 사용됩니다.

버전 3.7에 추가.

Task 객체

class asyncio.**Task**(*coro*, ***, *loop=None*)
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.

테스트를 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 True를 반환합니다.

*asyncio.Task*는 *Future.set_result()*와 *Future.set_exception()*을 제외한 모든 API를 *Future*에서 상속받습니다.

태스크는 *contextvars* 모듈을 지원합니다. 태스크가 만들어질 때 현재 컨텍스트를 복사하고 나중에 복사된 컨텍스트에서 코루틴을 실행합니다.

버전 3.7에서 변경: *contextvars* 모듈에 대한 지원이 추가되었습니다.

cancel()
Task 취소를 요청합니다.

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.

다음 예는 코루틴이 취소 요청을 가로채는 방법을 보여줍니다:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     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 취소*(cancelled)*됩니다.

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 None을 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.stderr*에 기록됩니다.

classmethod all_tasks(*loop=None*)
이벤트 루프의 모든 태스크 집합을 돌려줍니다.

기본적으로 현재 이벤트 루프에 대한 모든 태스크가 반환됩니다. *loop*가 None이면, 현재 루프를 가져오는 데 *get_event_loop()* 함수가 사용됩니다.

이 메서드는 **폐지되었고** 파이썬 3.9에서 제거됩니다. *asyncio.all_tasks()* 함수를 대신 사용하십시오.

classmethod current_task(*loop=None*)
현재 실행 중인 태스크나 None을 반환합니다.

*loop*가 None이면, 현재 루프를 가져오는 데 *get_event_loop()* 함수가 사용됩니다.

이 메서드는 **폐지되었고** 파이썬 3.9에서 제거됩니다. *asyncio.current_task()* 함수를 대신 사용하십시오.

제너레이터 기반 코루틴

참고: 제너레이터 기반 코루틴에 대한 지원은 **폐지되었고** 파이썬 3.10에서 삭제될 예정입니다.

제너레이터 기반 코루틴은 async/await 문법 전에 나왔습니다. 퓨처와 다른 코루틴을 기다리기 위해 yield from 표현식을 사용하는 파이썬 제너레이터입니다.

제너레이터 기반 코루틴은 *@asyncio.coroutine*으로 데코레이트 되어야 하지만 강제되지는 않습니다.

@asyncio.**coroutine**
제너레이터 기반 코루틴을 표시하는 데코레이터.

이 데코레이터는 기존 제너레이터 기반 코루틴이 async/await 코드와 호환되도록 합니다:

```
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

async def main():
    await old_style_coroutine()
```

이 데코레이터는 **폐지되었고** 파이썬 3.10에서 삭제 예정입니다.

async def 코루틴에는 이 데코레이터를 사용하면 안 됩니다.

asyncio.**iscoroutine**(*obj*)
*obj*가 코루틴 객체면 True를 반환합니다.

이 메서드는 제너레이터 기반 코루틴에 대해 True를 반환하기 때문에, *inspect.iscoroutine()*과 다릅니다.

asyncio.**iscoroutinefunction**(*func*)
*func*가 코루틴 함수면 True를 반환합니다.

이 메서드는 *@coroutine*으로 데코레이트 된 제너레이터 기반 코루틴 함수에 대해 True를 반환하기 때문에, *inspect.iscoroutinefunction()*과 다릅니다.

19.1.2 스트림

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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print(f'Send: {message!r}')
    writer.write(message.encode())

    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.:

coroutine asyncio.**open_connection**(*host=None*, *port=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

네트워크 연결을 만들고 `(reader, writer)` 객체 쌍을 반환합니다.

반환된 *reader* 와 *writer* 객체는 *StreamReader* 와 *StreamWriter* 클래스의 인스턴스입니다.

loop 인자는 선택적이며 이 함수를 코루틴에서 기다릴 때 언제나 자동으로 결정될 수 있습니다.

*limit*는 반환된 *StreamReader* 인스턴스가 사용하는 버퍼 크기 한계를 결정합니다. 기본적으로 *limit*는 64KB로 설정됩니다.

나머지 인자는 *loop.create_connection()*로 직접 전달됩니다.

버전 3.7에 추가: *ssl_handshake_timeout* 매개 변수.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, ***, *loop=None*, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_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.

loop 인자는 선택적이며, 이 메서드를 코루틴이 기다릴 때 항상 자동으로 결정될 수 있습니다.

*limit*는 반환된 *StreamReader* 인스턴스가 사용하는 버퍼 크기 한계를 결정합니다. 기본적으로 *limit*는 64KB로 설정됩니다.

나머지 인자는 *loop.create_server()*로 직접 전달됩니다.

버전 3.7에 추가: *ssl_handshake_timeout* 와 *start_serving* 매개 변수.

유닉스 소켓

coroutine asyncio.**open_unix_connection**(*path=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

유닉스 소켓 연결을 만들고 `(reader, writer)` 쌍을 반환합니다.

*open_connection()*과 비슷하지만, 유닉스 소켓에서 작동합니다.

*loop.create_unix_connection()*의 설명서도 참조하십시오.

가용성: 유닉스.

버전 3.7에 추가: *ssl_handshake_timeout* 매개 변수.

버전 3.7에서 변경: *path* 매개 변수는 이제 경로류 객체가 될 수 있습니다.

coroutine asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, ***, *loop=None*, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

유닉스 소켓 서버를 시작합니다.

*start_server()*와 비슷하지만, 유닉스 소켓에서 작동합니다.

*loop.create_unix_server()*의 설명서도 참조하십시오.

가용성: 유닉스.

버전 3.7에 추가: *ssl_handshake_timeout* 와 *start_serving* 매개 변수.

버전 3.7에서 변경: *path* 매개 변수는 이제 경로류 객체가 될 수 있습니다.

StreamReader

class asyncio.**StreamReader**

IO 스트림에서 데이터를 읽는 API를 제공하는 판독기(reader) 객체를 나타냅니다.

StreamReader 객체를 직접 인스턴스로 만드는 것은 권장되지 않습니다. 대신 *open_connection()*과 *start_server()*를 사용하십시오.

coroutine read(*n=-1*)

최대 *n* 바이트를 읽습니다. *n*이 제공되지 않거나 `-1`로 설정되면, EOF까지 읽은 후 모든 읽은 바이트를 반환합니다.

EOF를 수신했고 내부 버퍼가 비어 있으면, 빈 `bytes` 객체를 반환합니다.

coroutine 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.

coroutine readexactly(*n*)

정확히 *n* 바이트를 읽습니다.

n 바이트를 읽기 전에 EOF에 도달하면, *IncompleteReadError*를 일으킵니다. 부분적으로 읽은 데이터를 가져오려면 *IncompleteReadError.partial* 어트리뷰트를 사용하십시오.

coroutine 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.

버전 3.5.2에 추가.

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.

can_write_eof()
Return `True` if the underlying transport supports the *write_eof()* method, `False` otherwise.

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.

write(*data*)
스트림에 *data*를 기록합니다.

이 메서드는 흐름 제어의 대상이 아닙니다. `write()` 호출 뒤에는 *drain()*이 와야 합니다.

writelines(*data*)
스트림에 바이트열의 리스트(또는 임의의 이터러블)를 기록합니다.

이 메서드는 흐름 제어의 대상이 아닙니다. `writelines()` 호출 뒤에는 *drain()*이 와야 합니다.

coroutine 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.

close()
스트림을 닫습니다.

is_closing()
스트림이 닫혔거나 닫히고 있으면 `True`를 반환합니다.

버전 3.7에 추가.

coroutine wait_closed()
스트림이 닫힐 때까지 기다립니다.

하부 연결이 닫힐 때까지 기다리려면 *close()* 뒤에 호출해야 합니다.

버전 3.7에 추가.

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())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    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}')

    # Ignore the body, close the socket
    writer.close()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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

소켓이 *open_connection()* 함수를 사용하여 데이터를 수신할 때까지 기다리는 코루틴:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()

    # Close the second socket
    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.

19.1.3 동기화 프리미티브

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*

Lock

class asyncio.**Lock**(*, *loop=None*)

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()

# ... later
async with lock:
    # access shared state
```

이는 다음과 동등합니다:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

coroutine 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 True를 반환합니다.

Event

class asyncio.**Event**(*, *loop=None*)
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.

예:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

coroutine wait()
이벤트가 설정될 때까지 기다립니다.

이벤트가 설정되었으면 True를 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).

이제 *wait()*를 기다리는 태스크는 *set()* 메서드가 다시 호출될 때까지 블록 됩니다.

is_set()
이벤트가 설정되면 True를 반환합니다.

Condition

class asyncio.**Condition**(*lock=None*, *, *loop=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.

Condition을 사용하는 가장 좋은 방법은 async with 문입니다:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

이는 다음과 동등합니다:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

coroutine acquire()
하부 록을 얻습니다.

이 메서드는 하부 록이 풀림*(unlocked)*이 될 때까지 대기하고, 잠김*(locked)*으로 설정한 다음 True를 반환합니다.

notify(*n=1*)
이 조건을 기다리는 최대 *n* 태스크(기본적으로 1개)를 깨웁니다. 대기 중인 태스크가 없으면 이 메서드는 no-op입니다.

이 메서드를 호출하기 전에 록을 얻어야 하고, 호출 직후에 반납해야 합니다. 풀린*(unlocked)* 록으로 호출하면 *RuntimeError* 에러가 발생합니다.

locked()
하부 록을 얻었으면 True를 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.

coroutine 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 True를 돌려줍니다.

coroutine wait_for(*predicate*)
predicate가 참이 될 때까지 기다립니다.

predicate는 논릿값으로 해석될 결과를 돌려주는 콜러블이어야 합니다. 최종값이 반환 값입니다.

Semaphore

class asyncio.**Semaphore**(*value=1*, ***, *loop=None*)
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.

Semaphore를 사용하는 가장 좋은 방법은 async with 문입니다:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

이는 다음과 동등합니다:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()
세마포어를 얻습니다.

내부 카운터가 0보다 크면, 1 감소시키고 True를 즉시 반환합니다. 0이면, *release()*가 호출될 때까지 기다린 다음 True를 반환합니다.

locked()
세마포어를 즉시 얻을 수 없으면 True를 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*, ***, *loop=None*)
제한된 세마포어 객체. 스레드 안전하지 않습니다.

제한된 세마포어는 초기 *value* 위로 내부 카운터를 증가시키면 *release()*에서 *ValueError*를 발생시키는 *Semaphore* 버전입니다.

버전 3.7부터 폐지: `await lock` 이나 `yield from lock` 및/또는 `with` 문(`with await lock`, `with (yield from lock)`)을 사용하여 록을 얻는 것은 폐지되었습니다. 대신 `async with lock`을 사용하십시오.

19.1.4 서브 프로세스

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

coroutine asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

서브 프로세스를 만듭니다.

limit 인자는 `Process.stdout` 과 `Process.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.

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

cmd 셸 명령을 실행합니다.

limit 인자는 `Process.stdout` 과 `Process.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.

참고: 윈도우에서 기본 asyncio 이벤트 루프 구현은 서브 프로세스를 지원하지 않습니다. `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.

coroutine 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.

---

coroutine communicate(*input=None*)

프로세스와 상호 작용합니다:

1. 데이터를 *stdin*으로 보냅니다 (*input*이 `None`이 아니면);
2. EOF에 도달할 때까지 *stdout* 과 *stderr*에서 데이터를 읽습니다;
3. 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.

send_signal(*signal*)

시그널 *signal*를 자식 프로세스로 보냅니다.

> **참고:** 윈도우에서, SIGTERM은 *terminate()*의 별칭입니다. CTRL_C_EVENT 와 CTRL_BREAK_EVENT는 CREATE_NEW_PROCESS_GROUP을 포함하는 *creationflags* 매개 변수로 시작된 프로세스로 전송될 수 있습니다.

terminate()
자식 프로세스를 중지합니다.

POSIX 시스템에서 이 메서드는 *signal.SIGTERM*를 자식 프로세스로 보냅니다.

윈도우에서는 Win32 API 함수 TerminateProcess()가 호출되어 자식 프로세스를 중지합니다.

kill()
자식을 죽입니다.

POSIX 시스템에서 이 메서드는 SIGKILL를 자식 프로세스로 보냅니다.

윈도우에서 이 메서드는 *terminate()*의 별칭입니다.

stdin
표준 입력 스트림(*StreamWriter*) 또는 프로세스가 stdin=None으로 만들어졌으면 None.

stdout
표준 출력 스트림(*StreamReader*) 또는 프로세스가 stdout=None으로 만들어졌으면 None.

stderr
표준 에러 스트림(*StreamReader*) 또는 프로세스가 stderr=None으로 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 -N은 자식이 시그널 N으로 종료되었음을 나타냅니다 (POSIX만 해당).

서브 프로세스와 스레드

표준 asyncio 이벤트 루프는 다른 스레드에서 서브 프로세스를 실행하는 것을 지원하지만, 다음과 같은 제약이 있습니다:

- 이벤트 루프는 메인 스레드에서 실행되어야 합니다.
- 다른 스레드에서 서브 프로세스를 실행하기 전에 메인 스레드에서 자식 관찰자의 인스턴스를 만들어야 합니다. 메인 스레드에서 *get_child_watcher()* 함수를 호출하여 자식 감시자의 인스턴스를 만듭니다.

대체 이벤트 루프 구현은 위의 제한 사항을 공유하지 않을 수도 있습니다; 해당 설명서를 참조하십시오.

더 보기:

*asyncio*의 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())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

if sys.platform == "win32":
    asyncio.set_event_loop_policy(
        asyncio.WindowsProactorEventLoopPolicy())

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

저수준 API를 사용하여 작성된 같은 예제도 참조하십시오.

19.1.5 큐

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*, ***, *loop=None*)
선입 선출 (FIFO) 큐.

*maxsize*가 0보다 작거나 같으면 큐 크기는 무한합니다. 0보다 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.

이 클래스는 스레드 안전하지 않습니다.

maxsize
큐에 허용되는 항목 수.

empty()
큐가 비어 있으면 True를 반환하고, 그렇지 않으면 False를 반환합니다.

full()
큐에 *maxsize* 항목이 있으면 True를 반환합니다.

큐가 maxsize=0 (기본값)으로 초기화되었으면, *full()*은 절대 True를 반환하지 않습니다.

coroutine get()
큐에서 항목을 제거하고 반환합니다. 큐가 비어 있으면, 항목이 들어올 때까지 기다립니다.

get_nowait()
항목을 즉시 사용할 수 있으면 항목을 반환하고, 그렇지 않으면 *QueueEmpty*를 발생시킵니다.

coroutine 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.

coroutine put(*item*)
큐에 항목을 넣습니다. 큐가 가득 차면, 항목을 추가할 빈자리가 생길 때까지 기다립니다.

put_nowait(*item*)
블록하지 않고 항목을 큐에 넣습니다.

자리가 즉시 나지 않으면, *QueueFull*를 일으킵니다.

qsize()
큐에 있는 항목 수를 돌려줍니다.

task_done()
이전에 큐에 넣은 작업이 완료되었음을 나타냅니다.

큐 소비자가 사용합니다. 작업을 꺼내는 데 사용된 *get()* 마다, 뒤따르는 *task_done()* 호출은 작업에 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).

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.

예제

큐를 사용하여 여러 동시 태스크로 작업 부하를 분산시킬 수 있습니다:

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    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())
```

19.1.6 예외

exception asyncio.**TimeoutError**
작업이 주어진 마감 시간을 초과했습니다.

중요: 이 예외는 내장 *TimeoutError* 예외와 다릅니다.

---

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.

중요: 이 예외는 *Exception*의 서브 클래스이므로, 지나치게 광범위한 `try..except` 블록에 의해 의도하지 않게 억제될 수 있습니다:

```
try:
    await operation
except Exception:
    # The cancellation is broken because the *except* block
    # suppresses the CancelledError exception.
    log.log('an error has occurred')
```

대신, 다음 패턴을 사용해야 합니다:

```
try:
    await operation
except asyncio.CancelledError:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    raise
except Exception:
    log.log('an error has occurred')
```

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.

19.1.7 이벤트 루프

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*가 발생합니다. 이 함수는 코루틴이나 콜백에서만 호출할 수 있습니다.

버전 3.7에 추가.

asyncio.**get_event_loop**()
Get the current event loop.

If there is no current event loop set in the current OS thread, the OS thread is main, and *set_event_loop()* has not yet been called, asyncio will create a new event loop and set it as the current one.

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.

저수준 함수를 사용하여 수동으로 이벤트 루프를 만들고 닫는 대신 *asyncio.run()* 함수를 사용하는 것도 고려하십시오.

asyncio.**set_event_loop**(*loop*)
*loop*를 현재 OS 스레드의 현재 이벤트 루프로 설정합니다.

asyncio.**new_event_loop**()
새 이벤트 루프 객체를 만듭니다.

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*

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.

coroutine 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()
```

버전 3.6에 추가.

콜백 예약하기

loop.**call_soon**(*callback*, **args*, *context=None*)
이벤트 루프의 다음 이터레이션 때 *args* 인자로 호출할 *callback*을 예약합니다.

콜백은 등록된 순서대로 호출됩니다. 각 콜백은 정확히 한 번 호출됩니다.

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.

asyncio.Handle 인스턴스가 반환되는데, 나중에 콜백을 취소하는 데 사용할 수 있습니다.

이 메서드는 스레드 안전하지 않습니다.

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)
스레드 안전한 *call_soon()* 변형입니다. 다른 스레드에서 콜백을 예약하는 데 사용해야 합니다.

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:

```
# will schedule "print("Hello", flush=True)"
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.

버전 3.7에서 변경: *context* 키워드 전용 매개 변수가 추가되었습니다. 자세한 정보는 **PEP 567**을 보십시오.

버전 3.7.1에서 변경: 파이썬 3.7.0 및 이전 버전에서 기본 이벤트 루프 구현을 사용할 때, *delay*는 하루를 초과할 수 없었습니다. 이 문제는 파이썬 3.7.1에서 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.

버전 3.7에서 변경: *context* 키워드 전용 매개 변수가 추가되었습니다. 자세한 정보는 **PEP 567**을 보십시오.

버전 3.7.1에서 변경: 파이썬 3.7.0 및 이전 버전에서 기본 이벤트 루프 구현을 사용할 때, *when*와 현재 시각의 차이는 하루를 초과할 수 없었습니다. 이 문제는 파이썬 3.7.1에서 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 1일을 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

버전 3.5.2에 추가.

loop.**create_task**(*coro*)
코루틴 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.

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 콜러블 이어야 하고, (loop, coro) 과 일치하는 서명을 가져야 합니다. 여기서 *loop* 는 활성 이벤트 루프에 대한 참조가 되고, *coro* 는 코루틴 객체가 됩니다. 콜러블은 *asyncio.Future* 호환 객체를 반환해야 합니다.

loop.**get_task_factory**()
태스크 팩토리를 반환하거나, 기본값이 사용 중이면 None 을 반환합니다.

네트워크 연결 열기

coroutine 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*)

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.

- *sock* 이 주어지면, 트랜스포트가 사용할, 기존의 이미 연결된 *socket.socket* 객체여야 합니다. *sock* 이 주어지면, *host*, *port*, *family*, *proto*, *flags*, *local_addr* 를 지정해서는 안 됩니다.

- *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.

버전 3.7에 추가: *ssl_handshake_timeout* 매개 변수.

버전 3.6에서 변경: 소켓 옵션 `TCP_NODELAY`는 기본적으로 모든 TCP 연결에 대해 설정됩니다.

버전 3.5에서 변경: *ProactorEventLoop*에 SSL/TLS에 대한 지원이 추가되었습니다.

더 보기:

open_connection() 함수는 고수준 대안 API입니다. async/await 코드에서 직접 사용할 수 있는 (*StreamReader*, *StreamWriter*) 쌍을 반환합니다.

coroutine loop.**create_datagram_endpoint**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_address=None*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

참고: The parameter *reuse_address* is no longer supported, as using `SO_REUSEADDR` poses a significant security concern for UDP. Explicitly passing `reuse_address=True` will raise an exception.

When multiple processes with differing UIDs assign sockets to an indentical UDP socket address with `SO_REUSEADDR`, incoming packets can become randomly distributed among the sockets.

For supported platforms, *reuse_port* can be used as a replacement for similar functionality. With *reuse_port*, `SO_REUSEPORT` is used instead, which specifically prevents processes with differing UIDs from assigning sockets to the same socket address.

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.
- *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. `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).

윈도우에서, *ProactorEventLoop*를 사용할 때, 이 메서드는 지원되지 않습니다.

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 3.7.6에서 변경: The *reuse_address* parameter is no longer supported due to security concerns.

coroutine loop.**create_unix_connection**(*protocol_factory*, *path=None*, ***, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_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: 유닉스.

버전 3.7에 추가: *ssl_handshake_timeout* 매개 변수.

버전 3.7에서 변경: *path* 매개 변수는 이제 경로류 객체 가 될 수 있습니다.

네트워크 서버 만들기

coroutine 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*, *ssl_handshake_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).
- *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 `socket.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.
- *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.

- *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.
- *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 3.7에 추가: *ssl_handshake_timeout* 과 *start_serving* 매개 변수 추가.

버전 3.6에서 변경: 소켓 옵션 `TCP_NODELAY`는 기본적으로 모든 TCP 연결에 대해 설정됩니다.

버전 3.5에서 변경: *ProactorEventLoop*에 SSL/TLS에 대한 지원이 추가되었습니다.

버전 3.5.1에서 변경: *host* 매개 변수는 문자열의 시퀀스가 될 수 있습니다.

더 보기:

start_server() 함수는 async/await 코드에서 사용할 수 있는 *StreamReader* 및 *StreamWriter* 쌍을 반환하는 고수준의 대체 API입니다.

coroutine loop.**create_unix_server**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=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.

이 메서드의 인자에 대한 정보는 *loop.create_server()* 메서드의 설명서를 참조하십시오.

가용성: 유닉스.

버전 3.7에 추가: *ssl_handshake_timeout* 과 *start_serving* 매개 변수.

버전 3.7에서 변경: *path* 매개 변수는 이제 *Path* 객체일 수 있습니다.

coroutine loop.**connect_accepted_socket**(*protocol_factory*, *sock*, ***, *ssl=None*, *ssl_handshake_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.
- *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.

`(transport, protocol)` 쌍을 반환합니다.

버전 3.7에 추가: *ssl_handshake_timeout* 매개 변수.

버전 3.5.3에 추가.

파일 전송

coroutine 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.

버전 3.7에 추가.

TLS 업그레이드

coroutine loop.**start_tls**(*transport*, *protocol*, *sslcontext*, ***, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*)
기존 트랜스포트 기반 연결을 TLS로 업그레이드합니다.

protocol 이 *await* 의 직후에 사용해야 하는 새로운 트랜스포트 인스턴스를 반환합니다. *start_tls* 메서드에 전달된 *transport* 인스턴스는 절대로 다시 사용해서는 안 됩니다.

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.

버전 3.7에 추가.

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.

loop.**remove_reader**(*fd*)
fd 파일 기술자가 읽기 가능한지 관찰하는 것을 중단합니다.

loop.**add_writer**(*fd*, *callback*, **args*)
fd 파일 기술자가 쓰기 가능한지 관찰하기 시작하고, 일단 *fd* 가 쓰기 가능해지면 지정한 인자로 *callback* 을 호출합니다.

callback 에 키워드 인자를 전달하려면 *functools.partial()*를 사용하십시오.

loop.**remove_writer**(*fd*)
fd 파일 기술자가 쓰기 가능한지 관찰하는 것을 중단합니다.

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.

coroutine 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.

coroutine 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.

버전 3.7에 추가.

coroutine 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 *Future*를 반환했습니다. 파이썬 3.7부터, 이것은 `async def` 메서드입니다.

coroutine 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()*.

coroutine 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 바인드 된 주소입니다.

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.

더 보기:

loop.create_server() 와 *start_server()*.

coroutine 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.

버전 3.7에 추가.

DNS

coroutine loop.**getaddrinfo**(*host*, *port*, ***, *family=0*, *type=0*, *proto=0*, *flags=0*)
socket.getaddrinfo() 의 비동기 버전.

coroutine loop.**getnameinfo**(*sockaddr*, *flags=0*)
socket.getnameinfo() 의 비동기 버전.

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

coroutine 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.

coroutine 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 설정하는 데 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: 유닉스.

더 보기:

`signal` 모듈.

스레드 또는 프로세스 풀에서 코드를 실행하기

awaitable loop.**run_in_executor**(*executor*, *func*, **args*)
지정된 실행기에서 *func* 가 호출되도록 배치합니다.

executor 인자는 `Executor` 인스턴스여야 합니다. *executor* 가 `None` 이면 기본 실행기가 사용됩니다.

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))

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)

asyncio.run(main())
```

이 메서드는 *asyncio.Future* 객체를 반환합니다.

func 에 키워드 인자를 전달하려면 *functools.partial()*를 사용하십시오.

버전 3.5.3에서 변경: *loop.run_in_executor()* 는 더는 자신이 만드는 스레드 풀 실행기의 max_workers 를 설정하지 않습니다. 대신 스레드 풀 실행기(*ThreadPoolExecutor*)가 스스로 기본 값을 설정하도록 합니다.

loop.**set_default_executor**(*executor*)

executor 를 *run_in_executor()*에서 사용하는 기본 실행기로 설정합니다. *executor* 는 *ThreadPoolExecutor*의 인스턴스여야 합니다.

버전 3.7부터 폐지: *ThreadPoolExecutor* 인스턴스가 아닌 실행기의 사용은 폐지되었고, 파이썬 3.9에서는 에러를 일으키게 됩니다.

*executor*는 반드시 *concurrent.futures.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 (context에 대한 자세한 내용은 *`call_exception_handler()`* 문서를 참조하십시오).

`loop.`**`get_exception_handler`**()
현재 예외 처리기를 반환하거나, 사용자 정의 예외 처리기가 설정되지 않았으면 `None` 을 반환합니다.

버전 3.5.2에 추가.

`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;
- 'handle' (선택적): *`asyncio.Handle`* 인스턴스;
- 'protocol' (선택적): 프로토콜 인스턴스;
- 'transport' (선택적): 트랜스포트 인스턴스;
- 'socket' (선택적): *`socket.socket`* 인스턴스.

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 `-X dev` 명령 줄 옵션을 사용하여 디버그 모드를 활성화할 수 있습니다.

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: **Windows** 의 기본 asyncio 이벤트 루프는 자식 프로세스를 지원하지 않습니다. 자세한 내용은 윈도우에서의 자식 프로세스 지원을 참조하십시오.

---

coroutine 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*: `connect_write_pipe()`를 사용하여 자식 프로세스의 표준 입력 스트림에 연결될 파이프를 나타내는 파일류 객체 또는 `subprocess.PIPE` 상수 (기본값). 기본적으로 새 파이프가 만들어지고 연결됩니다.
- *stdout*: `connect_read_pipe()`를 사용하여 자식 프로세스의 표준 출력 스트림에 연결될 파이프를 나타내는 파일류 객체 또는 `subprocess.PIPE` 상수 (기본값). 기본적으로 새 파이프가 만들어지고 연결됩니다.
- *stderr*: `connect_read_pipe()`를 사용하여 자식 프로세스의 표준 에러 스트림에 연결될 파이프를 나타내는 파일류 객체 또는 `subprocess.PIPE` (기본값) 또는 `subprocess.STDOUT` 상수 중 하나.

  기본적으로 새 파이프가 만들어지고 연결됩니다. `subprocess.STDOUT`가 지정되면, 자식 프로세스의 표준 에러 스트림은 표준 출력 스트림과 같은 파이프에 연결됩니다.
- 다른 모든 키워드 인자는 해석 없이 `subprocess.Popen`로 전달됩니다. 다만, *bufsize*, *universal_newlines* 및 *shell*은 예외인데, 이것들은 지정되지 않아야 합니다.

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.

coroutine 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=True로 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.

cancel()

콜백을 취소합니다. 콜백이 이미 취소되었거나 실행되었다면 이 메서드는 아무 효과가 없습니다.

cancelled()

콜백이 취소되었으면 True 을 반환합니다.

버전 3.7에 추가.

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.

버전 3.7에 추가.

서버 객체

Server 객체는 *loop.create_server()*, *loop.create_unix_server()*, *start_server()*, *start_unix_server()*로 만듭니다.

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:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

버전 3.7에서 변경: Server 객체는 파이썬 3.7부터 비동기 컨텍스트 관리자입니다.

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 대기하려면 *wait_closed()* 코루틴을 사용하십시오.

get_loop()
서버 객체와 연관된 이벤트 루프를 반환합니다.

버전 3.7에 추가.

coroutine 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.

버전 3.7에 추가.

coroutine 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):
    # Communicate with the client with
    # reader/writer streams.  For example:
    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))
```

버전 3.7에 추가.

is_serving()
서버가 새 연결을 받아들이고 있으면 True 를 반환합니다.

버전 3.7에 추가.

coroutine wait_closed()
close() 메서드가 완료될 때까지 기다립니다.

sockets
서버가 리스닝하고 있는 *socket.socket* 객체의 리스트, 또는 서버가 닫혀 있다면 None.

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 모든 플랫폼에서 *SelectorEventLoop*를 사용하도록 구성됩니다.

class asyncio.**SelectorEventLoop**
selectors 모듈을 기반으로 하는 이벤트 루프.

주어진 플랫폼에서 사용할 수 있는 가장 효율적인 *selector*를 사용합니다. 정확한 셀렉터 구현을 수동으로 구성하여 사용할 수도 있습니다.:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

가용성: 유닉스, 윈도우.

class asyncio.**ProactorEventLoop**
"I/O 완료 포트"(IOCP)를 사용하는 윈도우용 이벤트 루프.

가용성: 윈도우.

윈도우에서 *ProactorEventLoop*를 사용하는 예:

```
import asyncio
import sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

더 보기:

I/O 완료 포트에 관한 MSDN 설명서.

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.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    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.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    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 SIGINT와 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)

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())
```

19.1.8 퓨처

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 True를 반환합니다:

- *asyncio.Future*의 인스턴스,
- *asyncio.Task*의 인스턴스,
- _asyncio_future_blocking 어트리뷰트를 가진 퓨처류 객체.

버전 3.5에 추가.

asyncio.**ensure_future**(*obj*, *, *loop=None*)
다음을 반환합니다:

- *obj*가 *Future*, *Task* 또는 퓨처류 객체면, *obj* 인자를 있는 그대로 (*isfuture()*로 검사합니다.)
- a *Task* object wrapping *obj*, if *obj* is a coroutine (*iscoroutine()* is used for the test); in this case the coroutine will be scheduled by ensure_future().
- *obj*가 어웨이터블이면, *obj*를 기다릴 *Task* 객체 (*inspect.isawaitable()*로 검사합니다.)

*obj*가 이 중 어느 것도 아니면, *TypeError*가 발생합니다.

중요: 새 태스크를 만드는 데 선호되는 *create_task()* 함수도 참조하십시오.

버전 3.5.1에서 변경: 함수는 모든 어웨이터블 객체를 받아들입니다.

asyncio.**wrap_future**(*future*, *, *loop=None*)
concurrent.futures.Future 객체를 *asyncio.Future* 객체로 감쌉니다.

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.

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.

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:

```
# Call 'print("Future:", fut)' when "fut" is done.
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()
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.

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.

버전 3.7에 추가.

이 예제는 Future 객체를 만들고, Future에 결과를 설정하는 비동기 Task를 만들고 예약하며, Future가 결과를 얻을 때까지 기다립니다:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    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 Future가 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 Future는 *concurrent.futures.wait()* 와 *concurrent.futures.as_completed()* 함수와 호환되지 않습니다.

19.1.9 트랜스포트와 프로토콜

머리말

트랜스포트와 프로토콜은 *loop.create_connection()* 와 같은 **저수준** 이벤트 루프 API에서 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/O와 프로세스 간 I/O를 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

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.

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.

버전 3.7에 추가.

`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.

버전 3.4.2에 추가.

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.

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: 표준 입력(*stdin*)의 읽을 수 있는 스트리밍 트랜스포트, 또는 서브 프로세스가 stdin=PIPE로 만들어지지 않았으면 *None*
- 1: 표준 출력(*stdout*)의 쓸 수 있는 스트리밍 트랜스포트, 또는 서브 프로세스가 stdout=PIPE로 만들어지지 않았으면 *None*
- 2: 표준 오류(*stderr*)의 쓸 수 있는 스트리밍 트랜스포트, 또는 서브 프로세스가 stderr=PIPE로 만들어지지 않았으면 *None*
- 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 SIGTERM을 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

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

버전 3.7에 추가: **중요:** 이것은 파이썬 3.7에서 잠정적으로 asyncio에 추가되었습니다! 이것은 실험용 API로서, 파이썬 3.8에서 변경되거나 완전히 제거될 수 있습니다.

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. errno가 *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.

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():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        lambda: EchoServerProtocol(),
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


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():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    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)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    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")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoServerProtocol(),
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    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
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    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

    def connection_made(self, transport):
        self.transport = transport
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

if sys.platform == "win32":
    asyncio.set_event_loop_policy(
        asyncio.WindowsProactorEventLoopPolicy())

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

고수준 API를 사용하여 작성된 같은 예제도 참조하십시오.

19.1.10 정책

이벤트 루프 정책은 이벤트 루프의 관리를 제어하는 프로세스별 전역 객체입니다. 각 이벤트 루프는 기본 정책이 있는데, 정책 API를 사용하여 변경하고 사용자 정의할 수 있습니다.

정책은 컨텍스트의 개념을 정의하고 컨텍스트마다 별도의 이벤트 루프를 관리합니다. 기본 정책은 컨텍스트를 현재 스레드로 정의합니다.

사용자 정의 이벤트 루프 정책을 사용하여, *get_event_loop()*, *set_event_loop()* 및 *new_event_loop()* 함수의 동작을 사용자 정의할 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.

asyncio.**set_event_loop_policy**(*policy*)
현재 프로세스 전반의 정책을 *policy*로 설정합니다.

*policy*를 None으로 설정하면, 기본 정책이 복원됩니다.

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 None을 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 None을 반환해서는 안 됩니다.

get_child_watcher()
자식 프로세스 감시자 객체를 얻습니다.

AbstractChildWatcher 인터페이스를 구현하고 있는 감시자 객체를 돌려줍니다.

이 함수는 유닉스 전용입니다.

set_child_watcher(*watcher*)
현재의 자식 프로세스 감시자를 *watcher*로 설정합니다.

이 함수는 유닉스 전용입니다.

asyncio에는 다음과 같은 내장 정책이 제공됩니다:

class asyncio.**DefaultEventLoopPolicy**
기본 asyncio 정책. 유닉스와 윈도우 플랫폼 모두에서 *SelectorEventLoop*를 사용합니다.

수동으로 기본 정책을 설치할 필요는 없습니다. asyncio는 기본 정책을 자동으로 사용하도록 구성됩니다.

class asyncio.**WindowsProactorEventLoopPolicy**
ProactorEventLoop 이벤트 루프 구현을 사용하는 대안 이벤트 루프 정책.

가용성: 윈도우.

프로세스 감시자

프로세스 감시자는 이벤트 루프가 유닉스에서 자식 프로세스를 관찰하는 방법을 사용자 정의할 수 있도록 합니다. 특히, 이벤트 루프는 자식 프로세스가 언제 종료했는지 알 필요가 있습니다.

asyncio에서, 자식 프로세스는 *create_subprocess_exec()* 와 *loop.subprocess_exec()* 함수로 만들어집니다.

asyncio는 자식 관찰자가 구현해야 하는 *AbstractChildWatcher* 추상 베이스 클래스를 정의하며, *SafeChildWatcher*(기본적으로 사용하도록 구성됩니다)와 *FastChildWatcher*의 두 가지 구현이 있습니다.

서브 프로세스와 스레드 절도 참조하십시오.

다음 두 함수를 사용하여 asyncio 이벤트 루프에서 사용되는 자식 프로세스 감시자 구현을 사용자 정의할 수 있습니다:

asyncio.**get_child_watcher**()
현재 정책에 대한 현재 자식 감시자를 반환합니다.

asyncio.**set_child_watcher**(*watcher*)
현재 정책에 대한 현재 자식 관찰자를 *watcher*로 설정합니다. *watcher*는 *AbstractChildWatcher* 베이스 클래스에 정의된 메서드를 구현해야 합니다.

참고: 제삼자 이벤트 루프 구현은 사용자 정의 자식 관찰자를 지원하지 않을 수 있습니다. 이러한 이벤트 루프에서는, *set_child_watcher()* 사용은 금지되거나 효과가 없습니다.

---

class asyncio.**AbstractChildWatcher**

add_child_handler(*pid*, *callback*, **args*)
새로운 자식 처리기를 등록합니다.

PID가 *pid* 인 프로세스가 종료할 때 `callback(pid, returncode, *args)` 가 호출되도록 배치합니다. 같은 프로세스에 대해 다른 콜백을 지정하면 이전 처리기가 교체됩니다.

callback 콜러블은 스레드 안전해야 합니다.

remove_child_handler(*pid*)
PID가 *pid* 인 프로세스의 처리기를 제거합니다.

이 함수는 처리기가 성공적으로 제거되면 `True`를, 제거할 것이 없으면 `False`를 반환합니다.

attach_loop(*loop*)
감시자를 이벤트 루프에 연결합니다.

감시자가 이전에 이벤트 루프에 연결되었으면, 새 루프에 연결하기 전에 먼저 제거됩니다.

참고: loop는 `None` 일 수 있습니다.

close()
감시자를 닫습니다.

이 메서드는 하부 자원을 정리하기 위해 호출해야 합니다.

class asyncio.**SafeChildWatcher**
이 구현은 SIGCHLD 시그널에 대해 명시적으로 모든 프로세스를 폴링하여 프로세스를 스포닝하는 다른 코드를 방해하지 않습니다.

이것은 안전한 해법이지만 많은 수의 프로세스를 처리할 때 상당한 오버헤드가 있습니다 (SIGCHLD가 수신될 때마다 *O(n)*).

asyncio는 기본적으로 이 안전한 구현을 사용합니다.

class asyncio.**FastChildWatcher**
이 구현은 os.waitpid(-1)를 직접 호출하여 종료된 모든 프로세스를 거둡니다. 프로세스를 스포닝하는 다른 코드를 망가뜨리고 그들의 종료를 기다릴 수 있습니다.

많은 수의 자식을 처리할 때 눈에 띄는 오버헤드가 없습니다 (자식이 종료될 때마다 *O(1)*).

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):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

19.1.11 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

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.create_datagram_endpoint()* 메서드는 지원되지 않습니다.
- *loop.add_reader()*와 *loop.add_writer()* 메서드는 지원되지 않습니다.

윈도우에서 단조 시계의 해상도는 대개 15.6 msec 근처입니다. 최상의 해상도는 0.5 msec입니다. 해상도는 하드웨어(HPET이 사용 가능한지)와 윈도우 구성에 따라 다릅니다.

윈도우에서의 서브 프로세스 지원

윈도우의 *SelectorEventLoop*는 서브 프로세스를 지원하지 않습니다. 윈도우에서는, 대신 *ProactorEventLoop*를 사용해야 합니다:

```
import asyncio

asyncio.set_event_loop_policy(
    asyncio.WindowsProactorEventLoopPolicy())

asyncio.run(your_code())
```

*ProactorEventLoop*가 자식 프로세스를 관찰하는 다른 메커니즘을 가지고 있으므로, *policy.set_child_watcher()* 함수도 지원되지 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)
```

19.1.12 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.
create_task()	asyncio Task를 시작합니다.
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.
current_task()	현재 Task를 돌려줍니다.
all_tasks()	이벤트 루프의 모든 태스크를 반환합니다.
Task	Task 객체.
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.

예제

- *asyncio.Event* 사용하기.
- asyncio 동기화 프리미티브의 설명서도 참조하십시오.

예외

asyncio.TimeoutError	*wait_for()*와 같은 함수에서 시간 초과 시 발생합니다. asyncio.TimeoutError는 내장 *TimeoutError* 예외와 **관련이 없음**에 유의하세요.
asyncio.CancelledError	Task가 취소될 때 발생합니다. *Task.cancel()*도 참조하십시오.

예제

- 취소 요청 시에 코드를 실행하기 위해 *CancelledError* 처리하기.
- *asyncio* 전용 예외의 전체 목록도 참조하십시오.

19.1.13 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 (현재 또는 정책을 통해).
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.

스레드/프로세스 풀

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_sendall()*	데이터를 *socket*으로 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.get_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 True를 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()	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 메서드**를 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.

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.

19.1.14 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.
- `-X dev` 파이썬 명령 줄 옵션 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:

- asyncio는 기다리지 않은 코루틴을 검사하고 로그 합니다; 이것은 "잊힌 await" 함정을 완화합니다.
- 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.
- 100ms보다 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)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

시그널을 처리하고 자식 프로세스를 실행하려면, 이벤트 루프를 메인 스레드에서 실행해야 합니다.

loop.run_in_executor() 메서드는 *concurrent.futures.ThreadPoolExecutor*와 함께 사용되어, 이벤트 루프가 실행되는 OS 스레드를 블록하지 않고 다른 OS 스레드에서 블로킹 코드를 실행할 수 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 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)
```

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")

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
```

19.2 socket — 저수준 네트워킹 인터페이스

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.

파이썬 인터페이스는 유닉스 시스템 호출과 소켓을 위한 라이브러리 인터페이스를 파이썬의 객체 지향 스타일로 직역한 것입니다: *socket()* 함수는 소켓 객체 *(socket object)*를 반환하고, 이것의 메서드는 다양한 소켓 시스템 호출을 구현합니다. 매개 변수 형은 C 인터페이스보다 다소 고수준입니다: 파이썬 파일에 대한 `read()` 와 `write()` 연산처럼, 수신 연산의 버퍼 할당은 자동이고 전송 연산에서 버퍼 길이는 묵시적입니다.

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.

- 쌍 `(host, port)`가 *AF_INET* 주소 패밀리에 사용됩니다. 여기서 *host*는 `'daring.cwi.nl'`과 같은 인터넷 도메인 표기법의 호스트 명이나 `'100.50.200.5'`와 같은 IPv4 주소를 나타내는 문자열이고, *port*는 정수입니다.

  - IPv4 주소의 경우, 호스트 주소 대신 두 개의 특수 형식이 허용됩니다: `''`는 모든 인터페이스에 바인딩하는 데 사용되는 INADDR_ANY를 나타내며 `'<broadcast>'` 문자열은 INADDR_BROADCAST를 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, scopeid)`가 사용됩니다. 여기서 *flowinfo* 와 *scopeid*는 C에서 `struct sockaddr_in6`의 `sin6_flowinfo` 와 `sin6_scope_id` 멤버를 나타냅니다. *socket* 모듈 메서드의 경우, *flowinfo* 와 *scopeid*는 이전 버전과의 호환성을 위해 생략할 수 있습니다. 그러나, *scopeid*를 생략하면 스코프가 지정된(scoped) IPv6 주소를 조작하는 데 문제가 발생할 수 있습니다.

  버전 3.7에서 변경: 멀티캐스트 주소(의미 있는 *scopeid*를 가진)의 경우, *address*에는 `%scope` (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.

- 문자열이나 튜플 `(id, unit)`는 `PF_SYSTEM` 패밀리의 `SYSPROTO_CONTROL` 프로토콜에 사용됩니다. 문자열은 동적으로 할당된 ID를 사용하는 커널 컨트롤의 이름입니다. 튜플은 커널 컨트롤의 ID와 유닛 번호가 알려져 있거나 등록된 ID가 사용될 때 사용할 수 있습니다.

  버전 3.3에 추가.

- `AF_BLUETOOTH`는 다음 프로토콜 및 주소 형식을 지원합니다:

  - `BTPROTO_L2CAP`는 `(bdaddr, psm)`를 받아들입니다. 여기서 `bdaddr`은 문자열 블루투스 주소이고 `psm`은 정수입니다.
  - `BTPROTO_RFCOMM`은 `(bdaddr, channel)`를 받아들입니다. 여기서 `bdaddr`은 문자열 블루투스 주소이고 `channel`은 정수입니다.
  - `BTPROTO_HCI`는 `(device_id,)`를 받아들입니다. 여기서 `device_id`는 정수나 인터페이스의 블루투스 주소인 문자열입니다. (이것은 여러분의 OS에 따라 다릅니다; NetBSD와 FreeBSD는 블루투스 주소를 기대하지만 다른 모든 것은 정수를 기대합니다.)

    버전 3.2에서 변경: NetBSD 및 DragonFlyBSD 지원이 추가되었습니다.

  - `BTPROTO_SCO`는 `bdaddr`를 받아들입니다. 여기서 `bdaddr`는 블루투스 주소의 문자열 형식이 포함된 *bytes* 객체입니다. (예, `b'12:23:34:45:56:67'`) 이 프로토콜은 FreeBSD에서 지원되지 않습니다.

- *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.

  Availability: Linux 2.6.38, some algorithm types require more recent Kernels.

  버전 3.6에 추가.

- *AF_VSOCK*은 가상 기계와 호스트가 통신할 수 있게 합니다. 소켓은 `(CID, port)` 튜플로 표현되는데, 컨텍스트 ID 또는 CID와 port는 정수입니다.

  Availability: Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

  버전 3.7에 추가.

- *AF_PACKET*은 네트워크 장치에 직접 연결된 저수준 인터페이스입니다. 패킷은 튜플 `(ifname, proto[, pkttype[, hatype[, addr]]])`로 표현됩니다. 여기서:

  - *ifname* - 장치 이름을 지정하는 문자열
  - *proto* - 이더넷 프로토콜 번호를 지정하는 네트워크 바이트 순서 정수.
  - *pkttype* - 패킷 유형을 지정하는 선택적 정수.:
    - `PACKET_HOST` (기본값) - 로컬 호스트로 향하는 패킷.
    - `PACKET_BROADCAST` - 물리 계층 브로드캐스트 패킷.
    - `PACKET_MULTIHOST` - 물리 계층 멀티캐스트 주소로 전송된 패킷.
    - `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.

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.

모든 에러는 예외를 발생시킵니다. 잘못된 인자 형과 메모리 부족 조건에 대한 일반적인 예외가 발생할 수 있습니다. 파이썬 3.3부터, 소켓이나 주소 의미와 관련된 에러는 *OSError* 나 그 서브 클래스 중 하나를 발생시킵니다 (예전에는 *socket.error*를 발생시켰습니다).

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**
*OSError*의 서브 클래스, 이 예외는 *getaddrinfo()* 와 *getnameinfo()*에 의한 주소 관련 에러에서 발생합니다. 수반되는 값은 라이브러리 호출이 반환한 에러를 나타내는 (error, string) 쌍입니다. *string*은 gai_strerror() C 함수가 반환한 *error*의 설명을 나타냅니다. 숫자 *error* 값은 이 모듈에 정의된 EAI_* 상수 중 하나와 일치합니다.

버전 3.3에서 변경: 이 클래스는 *OSError*의 서브 클래스가 되었습니다.

exception socket.**timeout**
*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.

상수

AF_* 와 SOCK_* 상수는 이제 AddressFamily 와 SocketKind *IntEnum* 컬렉션입니다.

버전 3.4에 추가.

socket.**AF_UNIX**
socket.**AF_INET**

socket.**AF_INET6**
이 상수는 *socket()*의 첫 번째 인자에 사용되는 주소 (및 프로토콜) 패밀리를 나타냅니다. *AF_UNIX* 상수가 정의되지 않으면 이 프로토콜은 지원되지 않습니다. 시스템에 따라 더 많은 상수를 사용할 수 있습니다.

socket.**SOCK_STREAM**
socket.**SOCK_DGRAM**
socket.**SOCK_RAW**
socket.**SOCK_RDM**
socket.**SOCK_SEQPACKET**
이 상수는 *socket()*의 두 번째 인자에 사용되는 소켓 유형을 나타냅니다. 시스템에 따라 더 많은 상수를 사용할 수 있습니다. (*SOCK_STREAM*과 *SOCK_DGRAM* 만 일반적으로 유용합니다.)

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:

좀 더 철저한 설명은 Secure File Descriptor Handling.

가용성: 리눅스 >= 2.6.27.

버전 3.2에 추가.

SO_*
socket.**SOMAXCONN**
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*
소켓 및/또는 IP 프로토콜에 대한 유닉스 설명서에서 설명된 이 형식의 많은 상수는 소켓 모듈에도 정의되어 있습니다. 일반적으로 소켓 객체의 `setsockopt()` 와 `getsockopt()` 메서드 인자에 사용됩니다. 대부분 유닉스 헤더 파일에 정의된 기호만 정의됩니다; 몇 가지 기호는 기본값이 제공됩니다.

버전 3.6에서 변경: `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION`가 추가되었습니다.

버전 3.6.5에서 변경: 윈도우에서, 런타임 윈도우가 지원하면 `TCP_FASTOPEN`, `TCP_KEEPCNT`가 나타납니다.

버전 3.7에서 변경: `TCP_NOTSENT_LOWAT`가 추가되었습니다.

윈도우에서, 런타임 윈도우가 지원하면 `TCP_KEEPIDLE`, `TCP_KEEPINTVL`가 나타납니다.

socket.**AF_CAN**
socket.**PF_CAN**
SOL_CAN_*
CAN_*
리눅스 설명서에 설명되어있는 이 형식의 많은 상수는 소켓 모듈에도 정의되어 있습니다.

가용성 : 리눅스 >= 2.6.25.

버전 3.3에 추가.

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 : 리눅스 >= 2.6.25.

버전 3.4에 추가.

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 : 리눅스 >= 3.6.

버전 3.5에 추가.

socket.**CAN_ISOTP**
CAN 프로토콜 패밀리의 CAN_ISOTP는 ISO-TP (ISO 15765-2) 프로토콜입니다. ISO-TP 상수는 리눅스 설명서에 설명되어 있습니다.

가용성 : 리눅스 >= 2.6.25.

버전 3.7에 추가.

socket.**AF_PACKET**
socket.**PF_PACKET**
PACKET_*
리눅스 설명서에 설명되어있는 이 형식의 많은 상수는 소켓 모듈에도 정의되어 있습니다.

가용성 : 리눅스 >= 2.2.

socket.**AF_RDS**
socket.**PF_RDS**
socket.**SOL_RDS**
RDS_*
리눅스 설명서에 설명되어있는 이 형식의 많은 상수는 소켓 모듈에도 정의되어 있습니다.

가용성 : 리눅스 >= 2.6.30.

버전 3.3에 추가.

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_PATH가 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 : 리눅스 >= 2.6.38.

버전 3.6에 추가.

socket.**AF_VSOCK**
socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**
VMADDR*
SO_VM*
리눅스 호스트/게스트 통신용 상수.

가용성 : 리눅스 >= 4.8.

버전 3.7에 추가.

socket.**AF_LINK**
가용성 : BSD, OSX.

버전 3.4에 추가.

socket.**has_ipv6**
이 상수는 이 플랫폼에서 IPv6가 지원되는지를 나타내는 논릿값을 포함합니다.

socket.**BDADDR_ANY**
socket.**BDADDR_LOCAL**
이들은 특수한 의미를 지닌 블루투스 주소를 포함하는 문자열 상수입니다. 예를 들어, *BDADDR_ANY*는 바인딩 소켓을 BTPROTO_RFCOMM로 지정할 때 임의의(any) 주소를 나타내는 데 사용할 수 있습니다.

socket.**HCI_FILTER**
socket.**HCI_TIME_STAMP**
socket.**HCI_DATA_DIR**
BTPROTO_HCI와 함께 사용하십시오. NetBSD 나 DragonFlyBSD에서는 *HCI_FILTER*를 사용할 수 없습니다. *HCI_TIME_STAMP*와 *HCI_DATA_DIR*는 FreeBSD, NetBSD 또는 DragonFlyBSD에서 사용할 수 없습니다.

함수

소켓 만들기

다음 함수는 모두 소켓 객체를 만듭니다.

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* 또는 *CAN_ISOTP* 중 하나여야 합니다.

*fileno*를 지정하면, *family*, *type* 및 *proto* 값이 지정된 파일 기술자에서 자동 감지됩니다. 명시적 *family*, *type* 또는 *proto* 인자를 사용하여 함수를 호출하면 자동 감지가 무효화 될 수 있습니다. 이는 파이썬이 *socket.getpeername()*의 반환 값을 나타내는 방식에 영향을 미치지만, 실제 OS 자원에는 영향을 주지 않습니다. *socket.fromfd()*와는 달리, *fileno*는 복제본이 아니라 같은 소켓을 반환합니다. 이렇게 하면 *socket.close()*를 사용하여 분리된 소켓을 닫을 수 있습니다.

새로 만들어진 소켓은 상속 불가능합니다.

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_NONBLOCK를 지원하는 OS에서 비 블로킹 소켓을 만들지만, sock.type은 socket.SOCK_STREAM로 설정됩니다.

socket.**socketpair**([*family*[, *type*[, *proto*]]])

제공된 주소 패밀리, 소켓 유형 및 프로토콜 번호를 사용하여 연결된 소켓 객체 쌍을 만듭니다. 주소 패밀리, 소켓 유형 및 프로토콜 번호는 위의 *socket()* 함수와 같습니다. 플랫폼에서 정의되어 있으면 기본 패밀리는 *AF_UNIX*입니다; 그렇지 않으면 기본값은 *AF_INET*입니다.

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*[, *source_address*]])

인터넷 *address*(2-튜플 (host, port))에서 리스닝하는 TCP 서비스에 연결하고 소켓 객체를 반환합니다. 이것은 *socket.connect()* 보다 고수준 함수입니다: *host*가 숫자가 아닌 호스트 명이면, *AF_INET*과 *AF_INET6* 모두로 결정하려고 시도한 다음, 연결이 성공할 때까지 차례대로 모든 가능한 주소로 연결을 시도합니다. 이것은 IPv4 및 IPv6 모두에 호환되는 클라이언트를 쉽게 작성할 수 있도록 합니다.

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.

버전 3.2에서 변경: *source_address*가 추가되었습니다.

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

파일 기술자 *fd*(파일 객체의 fileno() 메서드에서 반환된 정수)를 복제하고 결과로 소켓 객체를 만듭니다. 주소 패밀리, 소켓 유형 및 프로토콜 번호는 위의 *socket()* 함수와 같습니다. 파일 기술자는 소켓을 참조해야 하지만, 검사하지는 않습니다 — 파일 기술자가 유효하지 않으면 객체에 대한 후속 연산이 실패할 수 있습니다. 이 함수는 거의 필요하지 않지만, 프로그램에 표준 입력이나 출력으로 프로그램에 전달된 (가령 유닉스 inet 데몬으로 시작한 서버) 소켓의 소켓 옵션을 가져오거나 설정하는 데 사용할 수 있습니다. 소켓은 블로킹 모드로 간주합니다.

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: 윈도우.

버전 3.3에 추가.

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.

버전 3.7에 추가.

socket.**getaddrinfo**(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)
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.

family, *type* 및 *proto* 인자는 선택적으로 지정되어 반환된 주소 목록을 축소합니다. 이 인자 각각에 대한 값으로 0을 전달하면 전체 결과 범위가 선택됩니다. *flags* 인자는 `AI_*` 상수 중 하나 또는 여러 개일 수 있으며, 결과가 계산되고 반환되는 방식에 영향을 줍니다. 예를 들어, `AI_NUMERICHOST`는 도메인 이름 결정을 비활성화하고, *host*가 도메인 이름이면 에러를 발생시킵니다.

이 함수는 다음과 같은 구조의 5-튜플의 리스트를 반환합니다:

```
(family, type, proto, canonname, sockaddr)
```

이 튜플에서, *family*, *type*, *proto*는 모두 정수이며 *socket()* 함수로 전달됩니다. *canonname*은 `AI_CANONNAME`가 *flags* 인자의 일부일 때 *host*의 규범적(canonical) 이름을 나타내는 문자열입니다; 그렇지 않으면 *canonname*가 비어 있습니다. *sockaddr*은 반환된 *family*에 따라 형식이 달라지는, 소켓 주소를 설명하는 튜플이며 (*AF_INET*이면 `(address, port)` 2-튜플, *AF_INET6*이면 `(address, port, flow info, scope id)` 4-튜플), *socket.connect()* 메서드로 전달됩니다.

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)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 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` 부분이 포함되지 않습니다.

socket.**getfqdn**([*name*])
*name*의 완전히 정규화된 도메인 이름을 반환합니다. *name*이 생략되거나 비어 있으면, 지역 호스트로 해석됩니다. 완전히 정규화된 이름을 찾기 위해, *gethostbyaddr()*에 의해 반환된 호스트 이름이 검사되고, 있다면 그 호스트의 별칭이 뒤따릅니다. 마침표가 포함된 첫 번째 이름이 선택됩니다. 완전히 정규화된 도메인 이름이 없으면, *gethostname()*에서 반환된 호스트 이름이 반환됩니다.

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.

socket.**gethostbyname_ex**(*hostname*)
호스트 이름을 IPv4 주소 형식으로 변환합니다, 확장 인터페이스. 트리플 (hostname, aliaslist, ipaddrlist)를 반환합니다. 여기서 *hostname*은 지정된 *ip_address*에 응답하는 기본 호스트 이름이고, *aliaslist*는 같은 주소에 대한 대안 호스트 이름의 리스트(비어있을 수 있습니다)이며, *ipaddrlist*는 같은 호스트의 같은 인터페이스에 대한 IPv4 주소 리스트입니다 (항상 그렇지는 않지만, 종종 단일 주소). *gethostbyname_ex()*는 IPv6 이름 결정을 지원하지 않으며, IPv4/v6 이중 스택 지원을 위해서는 대신 *getaddrinfo()*를 사용해야 합니다.

socket.**gethostname**()
파이썬 인터프리터가 현재 실행 중인 기계의 호스트 명을 포함한 문자열을 반환합니다.

참고: *gethostname()*은 항상 완전히 정규화된 도메인 이름을 반환하지는 않습니다; 원한다면 *getfqdn()*을 사용하십시오.

socket.**gethostbyaddr**(*ip_address*)
트리플 (hostname, aliaslist, ipaddrlist)를 반환합니다. 여기서 *hostname*는 지정된 *ip_address*에 응답하는 기본 호스트 이름이고, *aliaslist*는 같은 주소에 대한 대체 호스트 이름의 (비어있을 수 있는) 리스트이며, *ipaddrlist*는 같은 호스트의 같은 인터페이스에 대한 IPv4/v6 주소 리스트입니다 (대개 단일 주소만 포함합니다). 완전히 정규화된 도메인 이름을 찾으려면, *getfqdn()* 함수를 사용하십시오. *gethostbyaddr()*는 IPv4와 IPv6를 모두 지원합니다.

socket.**getnameinfo**(*sockaddr*, *flags*)
소켓 주소 *sockaddr*를 2-튜플 (host, port)로 변환합니다. *flags*의 설정에 따라, 결과의 *host*에 완전히 정규화된 도메인 이름이나 숫자 주소 표현이 포함될 수 있습니다. 마찬가지로, *port*에는 문자열 포트 이름이나 숫자 포트 번호가 포함될 수 있습니다.

IPv6 주소의 경우, *sockaddr*에 의미 있는 *scopeid*가 있으면 %scope를 host 부분에 덧붙입니다. 보통 이것은 멀티캐스트 주소에서 일어납니다.

socket.**getprotobyname**(*protocolname*)
인터넷 프로토콜 이름(예를 들어, 'icmp')을 *socket()* 함수의 (선택적인) 세 번째 인자로 전달하기에 적합한 상수로 변환합니다. 이것은 일반적으로 "원시" 모드(*SOCK_RAW*)로 열린 소켓에만 필요합니다; 일반 소켓 모드에서는, 프로토콜이 생략되거나 0이면 올바른 프로토콜이 자동으로 선택됩니다.

socket.**getservbyname**(*servicename*[, *protocolname*])
인터넷 서비스 이름과 프로토콜 이름을 해당 서비스의 포트 번호로 변환합니다. 선택적 프로토콜 이름은, 주어진다면, 'tcp' 나 'udp' 여야 합니다, 그렇지 않으면 모든 프로토콜과 일치합니다.

socket.**getservbyport**(*port*[, *protocolname*])
인터넷 포트 번호와 프로토콜 이름을 해당 서비스의 서비스 이름으로 변환합니다. 선택적 프로토콜 이름은, 주어진다면, 'tcp' 나 'udp' 여야 합니다, 그렇지 않으면 모든 프로토콜과 일치합니다.

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 3.7부터 폐지: *x*가 16비트 부호 없는 정수에 맞지 않지만, 양의 C int에 맞으면, 16비트 부호 없는 정수로 자동 절단됩니다. 이 자동 절단 기능은 폐지되었으며, 미래 버전의 파이썬에서는 예외가 발생할 것입니다.

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 3.7부터 폐지: *x*가 16비트 부호 없는 정수에 맞지 않지만, 양의 C int에 맞으면, 16비트 부호 없는 정수로 자동 절단됩니다. 이 자동 절단 기능은 폐지되었으며, 미래 버전의 파이썬에서는 예외가 발생할 것입니다.

socket.**inet_aton**(*ip_string*)
IPv4 주소를 점 분리 쿼드 문자열 형식(예를 들어, '123.45.67.89')에서 길이가 4자인 바이트열 객체로 32비트 압축 바이너리 형식으로 변환합니다. 이 함수는 표준 C 라이브러리를 사용하고 `struct in_addr` 형(이 함수가 반환하는 32비트 압축 바이너리의 C형입니다)의 객체를 요구하는 프로그램과 대화할 때 유용합니다.

*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*)
32비트 압축 IPv4 주소(길이가 4바이트인 바이트열 객체)를 표준 점선 분리 쿼드 문자열 표현(예를 들어, '123.45.67.89')으로 변환합니다. 이 함수는 표준 C 라이브러리를 사용하고 `struct in_addr` 형(이 함수가 인자로 받아들이는 32비트 압축 바이너리 데이터의 C형입니다)의 객체를 요구하는 프로그램과 대화할 때 유용합니다.

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*)
패밀리 특정 문자열 형식의 IP 주소를 압축 바이너리 형식으로 변환합니다. *inet_pton()*는 라이브러리나 네트워크 프로토콜이 `struct in_addr` 형(*inet_aton()*과 유사)이나 `struct in6_addr` 형의 객체로 호출할 때 유용합니다.

*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: 유닉스(모든 플랫폼이 아닐 수도 있음), 윈도우.

버전 3.4에서 변경: 윈도우 지원이 추가되었습니다

socket.**inet_ntop**(*address_family*, *packed_ip*)
압축 IP 주소(일정 길이의 바이트열 객체)를 그것의 표준 패밀리 특정 문자열 표현(예를 들어, `'7.10.0.5'`나 `'5aef:2b::8'`)으로 변환합니다. *inet_ntop()*는 라이브러리나 네트워크 프로토콜이 `struct in_addr` 형(*inet_ntoa()*와 유사)이나 `struct in6_addr` 형의 객체를 반환할 때 유용합니다.

*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: 유닉스(모든 플랫폼이 아닐 수도 있음), 윈도우.

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: 대부분 유닉스 플랫폼, 다른 것들도 가능합니다.

버전 3.3에 추가.

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: 대부분 유닉스 플랫폼, 다른 것들도 가능합니다.

버전 3.3에 추가.

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 None입니다.

socket.**setdefaulttimeout**(*timeout*)
새 소켓 객체의 기본 시간제한을 초 단위로 (float) 설정합니다. 소켓 모듈을 처음 임포트 할 때 기본값은 None입니다. 가능한 값과 해당 의미는 *settimeout()*을 참조하십시오.

socket.**sethostname**(*name*)
기계의 호스트 명을 *name*으로 설정합니다. 충분한 권한이 없으면 *OSError*가 발생합니다.

가용성: 유닉스.

버전 3.3에 추가.

socket.**if_nameindex**()
네트워크 인터페이스 정보 (인덱스 정수, 이름 문자열) 튜플의 리스트를 반환합니다. 시스템 호출이 실패하면 *OSError*.

가용성: 유닉스.

버전 3.3에 추가.

socket.**if_nametoindex**(*if_name*)
인터페이스 이름에 대응하는 네트워크 인터페이스 인덱스 번호를 반환합니다. 주어진 이름을 가진 인터페이스가 없으면 *OSError*.

가용성: 유닉스.

버전 3.3에 추가.

socket.**if_indextoname**(*if_index*)
인터페이스 인덱스 번호에 해당하는 네트워크 인터페이스 이름을 반환합니다. 지정된 인덱스의 인터페이스가 없으면 *OSError*.

가용성: 유닉스.

버전 3.3에 추가.

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.)

socket.**close**()
소켓을 닫힌 상태로 표시합니다. 하부 시스템 자원(예를 들어, 파일 기술자)도 *makefile()*로 만든 모든 파일 객체가 닫힐 때 닫힙니다. 일단 닫히면, 소켓 객체에 대한 이후의 모든 연산이 실패합니다. 원격 끝은 더는 데이터를 수신하지 않게 됩니다 (계류 중인 데이터가 플러시 된 후에).

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()*는 연결과 관련된 자원을 해제하지만, 반드시 연결을 즉시 닫을 필요는 없습니다. 적시에 연결을 닫으려면, *close()* 전에 *shutdown()*을 호출하십시오.

---

socket.**connect**(*address*)
*address*에 있는 원격 소켓에 연결합니다. (*address*의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.)

시그널로 연결이 인터럽트 되면, 메서드는 연결이 완료될 때까지 대기하거나, 시그널 처리기가 예외를 발생시키지 않고 소켓이 블로킹하거나 시간제한이 있으면 *socket.timeout*을 발생시킵니다. 비 블로킹 소켓의 경우, 이 메서드는 시그널로 연결이 인터럽트 되면 *InterruptedError* 예외 (또는 시그널 처리기에서 발생한 예외)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).

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 0이고, 그렇지 않으면 errno 변수의 값입니다. 예를 들어 비동기 연결을 지원하는 데 유용합니다.

socket.**detach**()
하부 파일 기술자를 실제로 닫지 않으면서 소켓 객체를 닫힌 상태로 만듭니다. 파일 기술자가 반환되고, 다른 용도로 재사용 될 수 있습니다.

버전 3.2에 추가.

socket.**dup**()
소켓을 복제합니다.

새로 만들어진 소켓은 상속 불가능합니다.

버전 3.4에서 변경: 소켓은 이제 상속 불가능합니다.

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.

버전 3.4에 추가.

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*])
주어진 소켓 옵션의 값을 반환합니다 (유닉스 매뉴얼 페이지 *getsockopt(2)*를 보십시오). 필요한 기호 상수(SO_* 등)는 이 모듈에서 정의됩니다. *buflen*이 없으면, 정수 옵션을 가정하고 해당 정숫값이 함수에서 반환됩니다. *buflen*이 있으면, 옵션을 수신하는 데 사용되는 버퍼의 최대 길이를 지정하고, 이 버퍼가 바이트열 객체로 반환됩니다. 버퍼의 내용을 디코딩하는 것은 호출자의 책임입니다 (바이트열로 인코딩된 C 구조체를 디코딩하는 방법은 선택적 내장 모듈 *struct*를 참조하십시오).

socket.**getblocking**()
소켓이 블로킹 모드면 True를 반환하고, 비 블로킹이면 False를 반환합니다.

이것은 socket.gettimeout() == 0를 검사하는 것과 동등합니다.

버전 3.7에 추가.

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*)

플랫폼 윈도우

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.

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.

버전 3.5에서 변경: 이제 *backlog* 매개 변수가 선택적입니다.

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)
소켓과 결합한 파일 객체를 돌려줍니다. 정확한 반환형은 *makefile()* 에 주어진 인자에 따라 다릅니다. 이 인자는 내장 *open()* 함수와 같은 방식으로 해석됩니다. 단, 지원되는 *mode* 값은 'r' (기본값), 'w' 및 'b' 뿐입니다.

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*])
소켓에서 데이터를 수신합니다. 반환 값은 수신된 데이터를 나타내는 바이트열 객체입니다. 한 번에 수신할 수 있는 최대 데이터양은 *bufsize*에 의해 지정됩니다. 선택적 인자 *flags*의 의미는 유닉스 매뉴얼 페이지 *recv(2)* 를 보십시오; 기본값은 0입니다.

참고: 하드웨어와 네트워크 현실과 가장 잘 일치하려면, *bufsize*의 값은 2의 비교적 작은 거듭제곱이어야 합니다, 예를 들어 4096.

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 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 OR입니다; 자세한 내용은 시스템 설명서를 참조하십시오. 수신 소켓이 연결되어있지 않으면, *address*는 송신 소켓의 주소입니다, (사용 가능하다면); 그렇지 않으면 값은 지정되지 않습니다.

일부 시스템에서는, *sendmsg()*와 *recvmsg()*를 사용하여 *AF_UNIX* 소켓을 통해 프로세스 간에 파일 기술자를 전달할 수 있습니다. 이 기능을 사용하면(*SOCK_STREAM* 소켓으로 제한되는 경우가 많습니다), *recvmsg()*는 보조 데이터에서 `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)` 형식의 항목을 반환합니다. 여기서 *fds*는 새 파일 기술자를 네이티브 C `int` 형의 바이너리 배열로 나타내는 *bytes* 객체입니다. *recvmsg()*가 시스템 호출이 반환된 후에 예외를 발생시키면, 먼저 이 메커니즘을 통해 수신된 모든 파일 기술자를 닫으려고 시도합니다.

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: 대부분 유닉스 플랫폼, 다른 것들도 가능합니다.

버전 3.3에 추가.

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: 대부분 유닉스 플랫폼, 다른 것들도 가능합니다.

버전 3.3에 추가.

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 저장합니다. *nbytes*가 지정되지 않으면 (또는 0), 지정된 버퍼에서 사용 가능한 크기까지 수신합니다. 수신 한 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: 소켓 시간제한은 데이터가 성공적으로 전송될 때마다 더는 재설정되지 않습니다. 소켓 시간제한은 이제 모든 데이터를 전송할 수 있는 최대 총 지속 시간입니다.

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.)

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 None이 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: 대부분 유닉스 플랫폼, 다른 것들도 가능합니다.

버전 3.3에 추가.

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 : 리눅스 >= 2.6.38.

버전 3.6에 추가.

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.

버전 3.5에 추가.

socket.**set_inheritable**(*inheritable*)

소켓의 파일 기술자나 소켓 핸들의 상속 가능 플래그를 설정합니다.

버전 3.4에 추가.

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*)
블로킹 소켓 연산에 시간제한을 설정합니다. *value* 인자는 초로 표현된 음수가 아닌 부동 소수점 수나 None 일 수 있습니다. 0이 아닌 값을 주면, 후속 소켓 연산에서, 연산이 완료되기 전에 시간제한 기간 *value*가 지나면 *timeout* 예외를 발생시킵니다. 0을 지정하면, 소켓은 비 블로킹 모드가 됩니다. None이 주어지면, 소켓은 블로킹 모드가 됩니다.

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 the *socket* module (SO_* etc.). 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.

socket.**shutdown**(*how*)
연결의 한쪽 또는 양쪽 절반을 닫습니다. *how*가 SHUT_RD면, 추가 수신이 허용되지 않습니다. *how*가 SHUT_WR이면, 추가 전송이 허용되지 않습니다. *how*가 SHUT_RDWR이면, 추가 송수신이 허용되지 않습니다.

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: 윈도우.

버전 3.3에 추가.

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.
- 비 블로킹 모드에서, 연산은 즉시 완료할 수 없으면 실패합니다 (불행히도 시스템 종속적인 에러로): *select*의 함수를 사용하여 소켓이 읽기나 쓰기가 가능한 시기를 알 수 있습니다.
- 시간제한 모드에서, 연산은 소켓에 대해 지정된 제한 시간 내에 완료할 수 없거나 (*timeout* 예외 발생), 시스템이 에러를 반환하면 실패합니다.

참고: 운영 체제 수준에서, 시간제한 모드의 소켓은 내부적으로 비 블로킹 모드로 설정됩니다. 또한, 블로킹과 시간제한 모드는 같은 네트워크 끝점을 가리키는 파일 기술자와 소켓 객체 간에 공유됩니다. 이 구현 세부 사항은 가시적인 결과를 가져올 수 있습니다, 예를 들어, 소켓의 *fileno()*를 사용하기로 한 경우가 그렇습니다.

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

다음은 TCP/IP 프로토콜을 사용하는 4가지 최소 예제 프로그램입니다: (하나의 클라이언트만 서비스하는) 수신한 모든 데이터를 반향하는 서버와, 이를 사용하는 클라이언트. 서버는 *socket()*, *bind()*, *listen()*, *accept()* (두 개 이상의 클라이언트에 서비스를 제공하기 위해 *accept()*를 반복할 수 있습니다) 절차를 수행해야 하지만, 클라이언트는 *socket()*, *connect()* 절차만 요구함에 유의하십시오. 또한, 서버는 수신 대기 중인 소켓이 아니라 *accept()*가 반환한 새 소켓에 대해서 *sendall()*/*recv()*를 한다는 것에도 유의하십시오.

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))
```

다음 두 예제는 위의 두 예제와 같지만, IPv4와 IPv6를 모두 지원합니다. 서버 측은 사용 가능한 첫 번째 주소 패밀리를 리슨합니다(대신 두 주소를 모두 리슨 해야 합니다). 대부분 IPv6 지원 시스템에서, IPv6가 우선하며 서버가 IPv4 트래픽을 허용하지 않을 수 있습니다. 클라이언트 측은 이름 결정의 결과로 반환된 모든 주소에 연결을 시도하고 성공적으로 연결된 첫 번째 주소로 트래픽을 보냅니다.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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)
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 packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# receive a package
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)
```

소켓을 바인드(CAN_RAW)하거나 연결(*CAN_BCM*)한 후, *socket.send()* 와 *socket.recv()* 연산(과 대응 연산)을 소켓 객체에 평소와 같이 사용할 수 있습니다.

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)
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.

이것을 방지하기 위해서 설정할 수 있는 *socket* 플래그 socket.SO_REUSEADDR가 있습니다:

```
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 — 소켓 객체용 TLS/SSL 래퍼

소스 코드: Lib/ssl.py

이 모듈은 클라이언트 쪽과 서버 쪽 네트워크 소켓에 대한 전송 계층 보안(Transport Layer Security) ("보안 소켓 계층(Secure Sockets Layer)" 이라고도 함) 암호화와 피어 인증 기능에 대한 액세스를 제공합니다. 이 모듈은 OpenSSL 라이브러리를 사용합니다. OpenSSL이 해당 플랫폼에 설치되어있는 한, 모든 최신 유닉스 시스템, 윈도우, 맥 OS X 및 추가 플랫폼에서 사용할 수 있습니다.

참고: 운영 체제 소켓 API를 호출하기 때문에, 일부 동작은 플랫폼에 따라 다를 수 있습니다. 설치된 OpenSSL 버전도 동작을 바꿀 수 있습니다. 예를 들어, TLSv1.1과 TLSv1.2는 openssl 버전 1.0.1과 함께 제공됩니다.

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.

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, and cipher(), which retrieves the cipher being used for the secure connection.

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.

19.3.1 함수, 상수 및 예외

소켓 생성

파이썬 3.2와 2.7.9 이후로, *SSLSocket* 객체로 소켓을 포장하기 위해 *SSLContext* 인스턴스의 *SSLContext.wrap_socket()*을 사용하는 것이 좋습니다. 도우미 함수 *create_default_context()*는 보안 기본 설정의 새 컨텍스트를 반환합니다. 오래된 *wrap_socket()* 함수는 비효율적이고 서버 이름 표시(SNI)와 호스트 이름 일치를 지원하지 않기 때문에 폐지되었습니다.

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
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.

설정은: *PROTOCOL_TLS*, *OP_NO_SSLv2* 및 *OP_NO_SSLv3*이며, RC4가 없는 높은 암호화 사이퍼 스위트가 포함되고, 인증되지 않은 사이퍼 스위트는 포함되지 않습니다. *SERVER_AUTH*를 *purpose*로 전달하면 *verify_mode*가 *CERT_REQUIRED*로 설정되고 CA 인증서가 로드되거나 (*cafile*, *capath* 또는 *cadata* 중 하나 이상이 제공될 때), *SSLContext.load_default_certs()*를 사용하여 기본 CA 인증서를 로드합니다.

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 지원하는 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
```

버전 3.4에 추가.

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.

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 X509와 같이, 에러가 발생한 OpenSSL 하위 모듈을 지정하는 문자열 기호입니다. 가능한 값의 범위는 OpenSSL 버전에 따라 다릅니다.

버전 3.3에 추가.

reason
이 에러가 발생한 이유를 나타내는 문자열 기호, 예를 들어, `CERTIFICATE_VERIFY_FAILED`. 가능한 값의 범위는 OpenSSL 버전에 따라 다릅니다.

버전 3.3에 추가.

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.

버전 3.3에 추가.

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.

버전 3.3에 추가.

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.

버전 3.3에 추가.

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.

버전 3.3에 추가.

exception ssl.**SSLEOFError**
SSL 연결이 갑자기 종료되었을 때 발생하는 *SSLError*의 서브 클래스. 일반적으로, 이 에러가 발생하면 하부 트랜스포트를 다시 사용하지 않아야 합니다.

버전 3.3에 추가.

exception ssl.**SSLCertVerificationError**
인증서 유효성 검사가 실패했을 때 발생하는 *SSLError*의 서브 클래스.

버전 3.7에 추가.

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.

암호학적 생성기의 요구 사항을 얻으려면 위키피디아 기사 Cryptographically secure pseudorandom number generator (CSPRNG)를 읽으십시오.

버전 3.3에 추가.

ssl.**RAND_pseudo_bytes**(*num*)
(bytes, is_cryptographic) 을 반환합니다: bytes는 *num* 길이의 의사 난수 바이트열이며, 생성된 bytes가 암호학적으로 강하면 is_cryptographic은 `True`입니다. 현재 RAND 메서드에서 지원되지 않는 연산이면 *SSLError*를 발생시킵니다.

생성된 의사 난수 바이트 시퀀스는 충분한 길이일 때 고유하지만, 예측할 수 없는 것은 아닙니다. 이것들은 비암호화 목적이나 암호화 프로토콜에서의 특정 목적을 위해 사용될 수 있지만, 보통 키 생성 등을 위해 사용되지는 않습니다.

거의 모든 응용 프로그램에서 *os.urandom()*을 선호합니다.

버전 3.3에 추가.

버전 3.6부터 폐지: OpenSSL은 *ssl.RAND_pseudo_bytes()*를 폐지했습니다. 대신 *ssl.RAND_bytes()*를 사용하십시오.

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_egd**(*path*)
어딘가에 엔트로피 수집 데몬(EGD - entropy-gathering daemon)을 실행 중이고, *path*가 그곳으로 열려있는 소켓 연결의 경로명이면, 그 소켓에서 256바이트의 임의성을 읽고, 생성된 비밀 키의 보안을 강화하기 위해 이를 SSL 의사 난수 생성기에 추가합니다. 이것은 일반적으로 더 나은 임의성 소스가 없는 시스템에서만 필요합니다.

엔트로피 수집 데몬의 소스에 대해서는 http://egd.sourceforge.net/ 나 http://prngd.sourceforge.net/ 을 참조하십시오.

가용성: LibreSSL 과 OpenSSL > 1.1.0에서는 사용할 수 없습니다.

ssl.**RAND_add**(*bytes*, *entropy*)
주어진 *bytes*를 SSL 의사 난수 생성기에 섞습니다. 매개 변수 *entropy*(float)는 문자열에 포함된 엔트로피의 하한값이므로 항상 `0.0`을 사용할 수 있습니다. 엔트로피 소스에 대한 추가 정보는 **RFC 1750**을 참조하십시오.

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

인증서 처리

ssl.**match_hostname**(*cert*, *hostname*)

cert(*SSLSocket.getpeercert()*에서 반환된 디코딩된 형식)가 지정된 *hostname*과 일치하는지 확인합니다. 적용되는 규칙은 **RFC 2818**, **RFC 5280** 및 **RFC 6125**에 설명된 대로 HTTPS 서버의 신원(identity)을 확인하기 위한 것입니다. HTTPS 외에도, 이 함수는 FTPS, IMAPS, POPS 및 그 밖의 다양한 SSL 기반 프로토콜에서 서버의 신원을 확인하는 데 적합합니다.

실패하면 *CertificateError*가 발생합니다. 성공하면, 함수는 아무것도 반환하지 않습니다:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

버전 3.2에 추가.

버전 3.3.3에서 변경: 이 함수는 이제 **RFC 6125**, 6.4.3 절을 따르며 다중 와일드카드(예를 들어, `*.*.com` 나 `*a*.example.org`)나 국제화된 도메인 이름(IDN) 내부의 와일드카드와 일치하지 않습니다. `www*.xn--pthon-kva.org`와 같은 IDN A-레이블은 계속 지원되지만, `x*.python.org`는 더는 `xn--tda.python.org`와 일치하지 않습니다.

버전 3.5에서 변경: 인증서의 subjectAltName 필드에 있을 때, IP 주소의 일치는 이제 지원됩니다.

버전 3.7에서 변경: 이 함수는 더는 TLS 연결에 사용되지 않습니다. 이제 호스트 이름 일치는 OpenSSL에 의해 수행됩니다.

와일드카드가 가장 왼쪽에 있고 그 세그먼트의 유일한 문자일 때 와일드카드를 허용합니다. `www*.example.com`와 같은 부분적인 와일드카드는 더는 지원되지 않습니다.

버전 3.7부터 폐지.

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*, *ca_certs=None*)

주어진 SSL로 보호된 서버의 주소 `addr`((*hostname*, *port-number*) 쌍)에 대해, 서버 인증서를 가져와서 PEM-인코딩된 문자열로 반환합니다. `ssl_version`이 지정되면, 해당 버전의 SSL 프로토콜을 사용하여 서버에 연결을 시도합니다. `ca_certs`가 지정되면, 루트 인증서 목록을 포함하는 파일이어야 하는데, *SSLContext.wrap_socket()*에서 같은 매개 변수에 사용된 것과 같은 형식입니다. 호출은 해당

루트 인증서 집합에 대해 서버 인증서의 유효성을 검사하려고 시도하며, 유효성 검사 시도가 실패하면 실패합니다.

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.

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 DefaultVerifyPaths입니다.:

- 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

가용성: LibreSSL은 환경 변수 openssl_cafile_env와 openssl_capath_env를 무시합니다.

버전 3.4에 추가.

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: 윈도우.

버전 3.4에 추가.

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: 윈도우.

버전 3.4에 추가.

ssl.**wrap_socket**(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *ciphers=None*)

*socket.socket*의 인스턴스 sock을 취해서, SSL 컨텍스트에 하부 소켓을 감싸는 *socket.socket*의 서브 형인 *ssl.SSLSocket* 인스턴스를 반환합니다. sock은 *SOCK_STREAM* 소켓이어야합니다; 다른 소켓 유형은 지원되지 않습니다.

내부적으로, 함수는 프로토콜이 *ssl_version* 이고 *SSLContext.options*이 *cert_reqs*로 설정된 *SSLContext*를 만듭니다. 매개 변수 *keyfile*, *certfile*, *ca_certs* 또는 *ciphers*가 설정되면, 값은 *SSLContext.load_cert_chain()*, *SSLContext.load_verify_locations()* 및 *SSLContext.set_ciphers()*로 전달됩니다.

인자 *server_side*, *do_handshake_on_connect* 및 *suppress_ragged_eofs*는 *SSLContext.wrap_socket()*과 같은 의미입니다.

버전 3.7부터 폐지: 파이썬 3.2와 2.7.9부터, *wrap_socket()* 대신 *SSLContext.wrap_socket()*을 사용하는 것이 좋습니다. 최상위 함수는 제한적이고 서버 이름 표시나 호스트 이름 일치가 없는 안전하지 않은 클라이언트 소켓을 만듭니다.

상수

모든 상수는 이제 *enum.IntEnum* 이나 *enum.IntFlag* 컬렉션입니다.

버전 3.6에 추가.

ssl.**CERT_NONE**

*SSLContext.verify_mode*나 *wrap_socket()*의 cert_reqs 매개 변수의 가능한 값. *PROTOCOL_TLS_CLIENT*를 제외하고는, 기본 모드입니다. 클라이언트 측 소켓에서는, 모든 인증서가 허용됩니다. 신뢰할 수 없거나 만료된 인증서와 같은 유효성 검사 에러는 무시되며 TLS/SSL 핸드 셰이크를 중단하지 않습니다.

서버 모드에서는, 클라이언트에서 인증서를 요청하지 않으므로 클라이언트는 클라이언트 인증서 인증을 위해 인증서를 보내지 않습니다.

아래의 보안 고려 사항의 논의를 참조하십시오.

ssl.**CERT_OPTIONAL**

*SSLContext.verify_mode*나 *wrap_socket()*의 cert_reqs 매개 변수의 가능한 값. 클라이언트 모드에서, *CERT_OPTIONAL*는 *CERT_REQUIRED*와 같은 의미입니다. 클라이언트 측 소켓에서는 대신 *CERT_REQUIRED*를 사용하는 것이 좋습니다.

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 핸드 셰이크를 즉시 중단합니다.

이 설정을 사용하려면 유효한 CA 인증서 집합을 *SSLContext.load_verify_locations()* 나 *wrap_socket()*의 ca_certs 매개 변숫값으로 전달해야 합니다.

ssl.**CERT_REQUIRED**

*SSLContext.verify_mode*나 *wrap_socket()*의 cert_reqs 매개 변수의 가능한 값. 이 모드에서는, 소켓 연결의 다른 쪽에서 인증서를 요구합니다; 인증서가 제공되지 않거나 유효성 검사에 실패하면 *SSLError*가 발생합니다. 이 모드는 호스트 이름 일치를 수행하지 않기 때문에 클라이언트 모드에서 인증서를 유효성 검사하기에 충분하지 **않습니다**. 인증서의 진위를 검사하기 위해 *check_hostname*도 활성화해야 합니다. *PROTOCOL_TLS_CLIENT*는 기본적으로 *CERT_REQUIRED*를 사용하고 *check_hostname*을 활성화합니다.

서버 소켓에서, 이 모드는 필수 TLS 클라이언트 인증서 인증을 제공합니다. 클라이언트 인증서 요청이 클라이언트에 보내지고 클라이언트는 유효하고 신뢰할 수 있는 인증서를 제공해야 합니다.

이 설정을 사용하려면 유효한 CA 인증서 집합을 *SSLContext.load_verify_locations()* 나 *wrap_socket()*의 ca_certs 매개 변숫값으로 전달해야 합니다.

class ssl.**VerifyMode**
CERT_* 상수의 *enum.IntEnum* 컬렉션.

버전 3.6에 추가.

ssl.**VERIFY_DEFAULT**
*SSLContext.verify_flags*의 가능한 값. 이 모드에서는 인증서 해지 목록(CRL)을 검사하지 않습니다. 기본적으로 OpenSSL은 CRL을 요구하지도 검사하지도 않습니다.

버전 3.4에 추가.

ssl.**VERIFY_CRL_CHECK_LEAF**
Possible value for *SSLContext.verify_flags*. In this mode, only the peer cert is checked but none of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper CRL has has been loaded with *SSLContext.load_verify_locations*, validation will fail.

버전 3.4에 추가.

ssl.**VERIFY_CRL_CHECK_CHAIN**
*SSLContext.verify_flags*의 가능한 값. 이 모드에서는, 피어 인증서 체인의 모든 인증서에 대한 CRL이 확인됩니다.

버전 3.4에 추가.

ssl.**VERIFY_X509_STRICT**
망가진 X.509 인증서에 대한 우회를 사용하지 못하도록 하는 *SSLContext.verify_flags*의 가능한 값.

버전 3.4에 추가.

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.

버전 3.4.4에 추가.

class ssl.**VerifyFlags**
VERIFY_* 상수의 *enum.IntFlag* 컬렉션.

버전 3.6에 추가.

ssl.**PROTOCOL_TLS**
클라이언트와 서버가 모두 지원하는 가장 높은 프로토콜 버전을 선택합니다. 이름에도 불구하고, 이 옵션은 "SSL" 과 "TLS" 프로토콜을 모두 선택할 수 있습니다.

버전 3.6에 추가.

ssl.**PROTOCOL_TLS_CLIENT**
*PROTOCOL_TLS*처럼 가장 높은 프로토콜 버전을 자동 협상하지만, 클라이언트 측 *SSLSocket* 연결만 지원합니다. 이 프로토콜은 기본적으로 *CERT_REQUIRED*와 *check_hostname*을 활성화합니다.

버전 3.6에 추가.

ssl.**PROTOCOL_TLS_SERVER**
*PROTOCOL_TLS*처럼 가장 높은 프로토콜 버전을 자동 협상하지만, 서버 측 *SSLSocket* 연결만 지원합니다.

버전 3.6에 추가.

ssl.**PROTOCOL_SSLv23**
Alias for *PROTOCOL_TLS*.

버전 3.6부터 폐지: 대신 *PROTOCOL_TLS*를 사용하십시오.

ssl.**PROTOCOL_SSLv2**
채널 암호화 프로토콜로 SSL 버전 2를 선택합니다.

OpenSSL이 OPENSSL_NO_SSL2 플래그로 컴파일되었으면 이 프로토콜을 사용할 수 없습니다.

> **경고:** SSL 버전 2는 안전하지 않습니다. 사용하지 말도록 강력히 권고합니다.

버전 3.6부터 폐지: OpenSSL은 SSLv2에 대한 지원을 제거했습니다.

ssl.**PROTOCOL_SSLv3**
채널 암호화 프로토콜로 SSL 버전 3을 선택합니다.

OpenSSL이 OPENSSL_NO_SSLv3 플래그로 컴파일되었으면 이 프로토콜을 사용할 수 없습니다.

> **경고:** SSL 버전 3은 안전하지 않습니다. 사용하지 말도록 강력히 권고합니다.

버전 3.6부터 폐지: OpenSSL은 모든 버전 특정 프로토콜을 폐지했습니다. 대신 *OP_NO_SSLv3*와 같은 플래그와 함께 기본 프로토콜 *PROTOCOL_TLS*를 사용하십시오.

ssl.**PROTOCOL_TLSv1**
채널 암호화 프로토콜로 TLS 버전 1.0을 선택합니다.

버전 3.6부터 폐지: OpenSSL은 모든 버전 특정 프로토콜을 폐지했습니다. 대신 *OP_NO_SSLv3*와 같은 플래그와 함께 기본 프로토콜 *PROTOCOL_TLS*를 사용하십시오.

ssl.**PROTOCOL_TLSv1_1**
채널 암호화 프로토콜로 TLS 버전 1.1을 선택합니다. openssl 버전 1.0.1+ 에서만 사용할 수 있습니다.

버전 3.4에 추가.

버전 3.6부터 폐지: OpenSSL은 모든 버전 특정 프로토콜을 폐지했습니다. 대신 *OP_NO_SSLv3*와 같은 플래그와 함께 기본 프로토콜 *PROTOCOL_TLS*를 사용하십시오.

ssl.**PROTOCOL_TLSv1_2**
채널 암호화 프로토콜로 TLS 버전 1.2를 선택합니다. 이것은 가장 현대적인 버전이며, 양측이 모두 가능하다면 최대한의 보호를 위해 아마도 제일 나은 선택입니다. openssl 버전 1.0.1+ 에서만 사용할 수 있습니다.

버전 3.4에 추가.

버전 3.6부터 폐지: OpenSSL은 모든 버전 특정 프로토콜을 폐지했습니다. 대신 *OP_NO_SSLv3*와 같은 플래그와 함께 기본 프로토콜 *PROTOCOL_TLS*를 사용하십시오.

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.

버전 3.2에 추가.

ssl.**OP_NO_SSLv2**
SSLv2 연결을 방지합니다. 이 옵션은 *PROTOCOL_TLS*와 결합해서만 적용할 수 있습니다. 피어가 SSLv2를 프로토콜 버전으로 선택하지 못하도록 합니다.

버전 3.2에 추가.

버전 3.6부터 폐지: 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.

버전 3.2에 추가.

버전 3.6부터 폐지: 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.

버전 3.2에 추가.

버전 3.7부터 폐지: 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.

버전 3.4에 추가.

버전 3.7부터 폐지: 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.

버전 3.4에 추가.

버전 3.7부터 폐지: 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.

버전 3.7에 추가.

버전 3.7부터 폐지: 이 옵션은 OpenSSL 1.1.0부터 폐지되었습니다. OpenSSL 1.0.2와의 하위 호환성을 위해 2.7.15, 3.6.3 및 3.7.0에 추가되었습니다.

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.

버전 3.7에 추가.

ssl.**OP_CIPHER_SERVER_PREFERENCE**
클라이언트보다는 서버의 사이퍼 순서 선호를 사용합니다. 이 옵션은 클라이언트 소켓과 SSLv2 서버 소켓에는 영향을 미치지 않습니다.

버전 3.3에 추가.

ssl.**OP_SINGLE_DH_USE**
서로 다른 SSL 세션에 대해 같은 DH 키 재사용을 방지합니다. 이렇게 하면 FS(forward secrecy)는 향상되지만, 더 많은 계산 자원을 요구합니다. 이 옵션은 서버 소켓에만 적용됩니다.

버전 3.3에 추가.

ssl.**OP_SINGLE_ECDH_USE**
서로 다른 SSL 세션에 대해 같은 ECDH 키 재사용을 방지합니다. 이렇게 하면 FS(forward secrecy)는 향상되지만, 더 많은 계산 자원을 요구합니다. 이 옵션은 서버 소켓에만 적용됩니다.

버전 3.3에 추가.

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.

버전 3.8에 추가.

ssl.**OP_NO_COMPRESSION**
SSL 채널에서 압축을 사용하지 않습니다. 응용 프로그램 프로토콜이 자체 압축 방법을 지원할 때 유용합니다.

이 옵션은 OpenSSL 1.0.0 이상에서만 사용할 수 있습니다.

버전 3.3에 추가.

class ssl.**Options**
OP_* 상수의 *enum.IntFlag* 컬렉션.

ssl.**OP_NO_TICKET**
클라이언트 측에서 세션 티켓을 요청하지 못하게 합니다.

버전 3.6에 추가.

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

버전 3.5에 추가.

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.

버전 3.7에 추가.

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.

버전 3.3에 추가.

ssl.**HAS_SNI**
OpenSSL 라이브러리가 서버 이름 표시*(Server Name Indication)* 확장(**RFC 6066**에 정의된 대로)에 대한 지원을 기본 제공하는지 여부.

버전 3.2에 추가.

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.

버전 3.3에 추가.

ssl.**HAS_SSLv2**
OpenSSL 라이브러리가 SSL 2.0 프로토콜 지원을 기본 제공하는지 여부

버전 3.7에 추가.

ssl.**HAS_SSLv3**
OpenSSL 라이브러리가 SSL 3.0 프로토콜 지원을 기본 제공하는지 여부

버전 3.7에 추가.

ssl.**HAS_TLSv1**
OpenSSL 라이브러리가 TLS 1.0 프로토콜 지원을 기본 제공하는지 여부

버전 3.7에 추가.

ssl.**HAS_TLSv1_1**
OpenSSL 라이브러리가 TLS 1.1 프로토콜 지원을 기본 제공하는지 여부

버전 3.7에 추가.

ssl.**HAS_TLSv1_2**
OpenSSL 라이브러리가 TLS 1.2 프로토콜 지원을 기본 제공하는지 여부

버전 3.7에 추가.

ssl.**HAS_TLSv1_3**
OpenSSL 라이브러리가 TLS 1.3 프로토콜 지원을 기본 제공하는지 여부

버전 3.7에 추가.

ssl.**CHANNEL_BINDING_TYPES**
지원되는 TLS 채널 바인딩 유형의 리스트. 이 리스트의 문자열은 *SSLSocket.get_channel_binding()*에 대한 인자로 사용될 수 있습니다.

버전 3.3에 추가.

ssl.**OPENSSL_VERSION**
인터프리터에 의해 로드된 OpenSSL 라이브러리의 버전 문자열:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

버전 3.2에 추가.

ssl.**OPENSSL_VERSION_INFO**
OpenSSL 라이브러리에 대한 버전 정보를 나타내는 5개의 정수 튜플:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

버전 3.2에 추가.

ssl.**OPENSSL_VERSION_NUMBER**
단일 정수로 표현되는, OpenSSL 라이브러리의 원시 버전 번호:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

버전 3.2에 추가.

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.

버전 3.4에 추가.

class ssl.**AlertDescription**
ALERT_DESCRIPTION_* 상수의 *enum.IntEnum* 컬렉션.

버전 3.6에 추가.

Purpose.**SERVER_AUTH**
*create_default_context()*와 *SSLContext.load_default_certs()* 옵션. 이 값은 컨텍스트가 웹 서버를 인증하는 데 사용될 수 있음을 나타냅니다 (따라서 클라이언트 측 소켓을 만드는 데 사용됩니다).

버전 3.4에 추가.

Purpose.**CLIENT_AUTH**
create_default_context() 와 *SSLContext.load_default_certs()* 옵션. 이 값은 컨텍스트가 웹 클라이언트를 인증하는 데 사용될 수 있음을 나타냅니다 (따라서 서버 측 소켓을 만드는 데 사용됩니다).

버전 3.4에 추가.

class ssl.**SSLErrorNumber**
SSL_ERROR_* 상수의 *enum.IntEnum* 컬렉션.

버전 3.6에 추가.

class ssl.**TLSVersion**
*SSLContext.maximum_version*과 *SSLContext.minimum_version* 용 SSL과 TLS 버전의 *enum.IntEnum* 컬렉션.

버전 3.7에 추가.

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.

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()* (그러나 0이 아닌 flags 인자를 전달하는 것은 허용되지 않습니다)
- *send()*, *sendall()* (같은 제한 있음)
- *sendfile()* (그러나 *os.sendfile*는 평문 소켓에만 사용되며, 그렇지 않으면 *send()*가 사용됩니다)
- *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: shutdown()은 바이트가 수신되거나 전송될 때마다 소켓 시간제한을 재설정하지 않습니다. 소켓 시간제한은 이제 shutdown의 최대 총 지속 시간입니다.

버전 3.6부터 폐지: *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.

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: 소켓 시간제한은 바이트가 수신되거나 전송될 때마다 재설정되지 않습니다. 소켓 시간제한은 이제 최대 *len* 바이트까지 읽을 때까지의 최대 총 지속 시간입니다.

버전 3.6부터 폐지: *read()* 대신 recv()를 사용하십시오.

SSLSocket.**write**(*buf*)

SSL 소켓에 *buf*를 기록하고, 기록한 바이트 수를 돌려줍니다. *buf* 인자는 버퍼 인터페이스를 지원하는 객체여야 합니다.

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: 소켓 시간제한은 바이트가 수신되거나 전송될 때마다 재설정되지 않습니다. 소켓 시간제한은 이제 *buf*를 쓰는 최대 총 지속 시간입니다.

버전 3.6부터 폐지: *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**()

SSL 설정 핸드 셰이크를 수행합니다.

버전 3.4에서 변경: 핸드 셰이크 메서드는 소켓의 *context*의 *check_hostname* 어트리뷰트가 참일 때 *match_hostname()*도 수행합니다.

버전 3.5에서 변경: 소켓 시간제한은 바이트가 수신되거나 전송될 때마다 재설정되지 않습니다. 소켓 시간제한은 이제 핸드 셰이크의 최대 지속 시간입니다.

버전 3.7에서 변경: 호스트 이름이나 IP 주소는 핸드 셰이크 중 OpenSSL에서 일치합니다. 함수 *match_hostname()*는 더는 사용되지 않습니다. OpenSSL이 호스트 이름이나 IP 주소를 거절할 경우, 핸드 셰이크가 일찍 중단되고 TLS 경고 메시지가 상대방에게 전송됩니다.

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}
```

참고: 특정 서비스를 위해 인증서를 유효성 검사하려면, *match_hostname()* 함수를 사용할 수 있습니다.

---

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.

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 3.7.6에서 변경: IPv6 address strings no longer have a trailing new line.

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**()
핸드 셰이크 중에 클라이언트에 의해 공유되는 사이퍼의 리스트를 돌려줍니다. 반환된 리스트의 각 항목은 사이퍼의 이름, 그것의 사용을 정의하는 SSL 프로토콜의 버전 및 사이퍼가 사용하는 비밀 비트의 수를 포함하는 3-튜플입니다. 연결이 이루어지지 않았거나 소켓이 클라이언트 소켓이면 *shared_ciphers()*는 None을 반환합니다.

버전 3.5에 추가.

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.

버전 3.3에 추가.

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.

버전 3.3에 추가.

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.

버전 3.5에 추가.

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.

버전 3.3에 추가.

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.

버전 3.7.1에 추가.

SSLSocket.**version**()
연결에서 협상한 실제 SSL 프로토콜 버전을 문자열로 반환하거나, 보안 연결이 이루어지지 않았으면 None을 반환합니다. 이 글을 쓰는 시점에서, 가능한 반환 값은 "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" 및 "TLSv1.2"입니다. 최근의 OpenSSL 버전은 더욱더 많은 반환 값을 정의할 수 있습니다.

버전 3.5에 추가.

SSLSocket.**pending**()
접속에 계류 중인, 읽기용으로 이미 복호화된 바이트의 수를 돌려줍니다.

SSLSocket.**context**
이 SSL 소켓이 연결된 *SSLContext* 객체. SSL 소켓이 폐지된 *wrap_socket()* 함수(*SSLContext.wrap_socket()*이 아니라)를 사용하여 만들어졌으면, 이것은 이 SSL 소켓에 대해 만든 사용자 정의 컨텍스트 객체입니다.

버전 3.2에 추가.

SSLSocket.**server_side**
서버 측 소켓에서는 True이고 클라이언트 측 소켓에서는 False 인 논릿값.

버전 3.2에 추가.

SSLSocket.**server_hostname**
서버의 호스트 이름: *str* 형, 또는 서버 측 소켓이거나 호스트 이름이 생성자에 지정되지 않았으면 None.

버전 3.2에 추가.

버전 3.7에서 변경: 어트리뷰트는 이제 항상 ASCII 텍스트입니다. server_hostname이 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.

버전 3.6에 추가.

SSLSocket.**session_reused**
버전 3.6에 추가.

19.3.3 SSL 컨텍스트

버전 3.2에 추가.

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=PROTOCOL_TLS*)

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: 컨텍스트는 안전한 기본값으로 생성됩니다. *OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* (*PROTOCOL_SSLv2* 제외) 및 *OP_NO_SSLv3* (*PROTOCOL_SSLv3* 제외) 옵션은 기본적으로 설정됩니다. 초기 사이퍼 스위트 리스트에는 HIGH 사이퍼만 포함되고 NULL 사이퍼나 MD5 사이퍼는 포함되지 않습니다 (*PROTOCOL_SSLv2* 제외).

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}
```

버전 3.4에 추가.

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)

개인 키와 해당 인증서를 로드합니다. *certfile* 문자열은 인증서뿐만 아니라 인증서의 신뢰성을 확보하는 데 필요한 여러 CA 인증서가 포함된 PEM 형식의 단일 파일 경로여야 합니다. *keyfile* 문자열이 있으면 개인 키가 들어있는 파일을 가리켜야 합니다. 그렇지 않으면 개인 키도 *certfile*에서 가져옵니다. 인증서가 *certfile*에 저장되는 방법에 대한 자세한 내용은 인증서의 논의를 참조하십시오.

[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.

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*)

기본 위치에서 기본 "인증 기관"(CA) 인증서 집합을 로드합니다. 윈도우에서는 CA 와 ROOT 시스템 저장소에서 CA 인증서를 로드합니다. 다른 시스템에서는 *SSLContext.set_default_verify_paths()*를 호출합니다. 장래에 이 메서드는 다른 위치에서도 CA 인증서를 로드할 수 있습니다.

purpose 플래그는 로드되는 CA 인증서의 종류를 지정합니다. 기본 설정인 *Purpose.SERVER_AUTH*는 TLS 웹 서버 인증 (클라이언트 측 소켓)용으로 표시되고 신뢰 되는 인증서를 로드합니다. *Purpose.CLIENT_AUTH*는 서버 측에서 클라이언트 인증서 확인을 위한 CA 인증서를 로드합니다.

버전 3.4에 추가.

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.

capath 문자열이 있으면, OpenSSL 특정 배치를 따르는 PEM 형식의 여러 CA 인증서를 포함하는 디렉터리에 대한 경로입니다.

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.

버전 3.4에 추가.

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()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

OpenSSL 1.1 이상에서 사이퍼 딕셔너리에는 다음과 같은 추가 필드가 포함됩니다:

```
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'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'}]
```

가용성: OpenSSL 1.0.2+.

버전 3.6에 추가.

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*)

이 컨텍스트로 만들어진 소켓에 사용할 수 있는 사이퍼를 설정합니다. OpenSSL 사이퍼 리스트 형식의

문자열이어야 합니다. 사이퍼를 아무것도 선택할 수 없으면 (컴파일 시간 옵션이나 다른 구성이 지정된 모든 사이퍼의 사용을 금지하기 때문에), *SSLError*가 발생합니다.

참고: 연결될 때, SSL 소켓의 *SSLSocket.cipher()* 메서드가 현재 선택된 사이퍼를 제공합니다.

OpenSSL 1.1.1 에는 기본적으로 활성화된 TLS 1.3 사이퍼 스위트가 있습니다. 이 스위트는 *set_ciphers()*로 비활성화할 수 없습니다.

SSLContext.**set_alpn_protocols**(*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.

This method will raise *NotImplementedError* if *HAS_ALPN* is `False`.

OpenSSL 1.1.0 에서 1.1.0e 는 양측이 ALPN 을 지원하지만, 프로토콜에 합의할 수 없으면 핸드 셰이크를 중지하고 *SSLError*를 발생시킵니다. 1.1.0f+ 는 1.0.2 처럼 동작합니다, *SSLSocket.selected_alpn_protocol()*는 None 을 반환합니다.

버전 3.5 에 추가.

SSLContext.**set_npn_protocols**(*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.

This method will raise *NotImplementedError* if *HAS_NPN* is `False`.

버전 3.3 에 추가.

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.

TLS 연결의 초기 협상 단계로 인해, *SSLSocket.selected_alpn_protocol()*과 *SSLSocket.context*와 같은 제한된 메서드와 어트리뷰트만 사용할 수 있습니다. *SSLSocket.getpeercert()*, *SSLSocket.getpeercert()*, *SSLSocket.cipher()* 및 `SSLSocket.compress()` 메서드는 TLS 연결이 TLS 클라이언트 Hello 이상으로 진행되었을 것을 요구하므로, 의미 있는 값을 반환하거나 안전하게 호출할 수 없습니다.

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.

버전 3.7에 추가.

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.

서버 이름에 디코딩 에러가 있으면, TLS 연결이 클라이언트로 *ALERT_DESCRIPTION_INTERNAL_ERROR* 치명적인 TLS 경고 메시지를 주면서 종료됩니다.

버전 3.4에 추가.

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.

버전 3.3에 추가.

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 False면 이 메서드를 사용할 수 없습니다.

버전 3.3에 추가.

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*)
기존 파이썬 소켓 *sock*을 감싸고 *SSLContext.sslsocket_class*(기본값 *SSLSocket*)의 인스턴스를 반환합니다. 반환 된 SSL 소켓은 컨텍스트, 설정 및 인증서에 연결됩니다. *sock*은 *SOCK_STREAM* 소켓이어야합니다; 다른 소켓 유형은 지원되지 않습니다.

매개 변수 server_side는 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_connect는 socket.connect()를 수행한 후 SSL 핸드 셰이크를 자동으로 수행할지, 또는 응용 프로그램이 *SSLSocket.do_handshake()* 메서드를 호출하여 명시적으

로 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.

버전 3.5에서 변경: OpenSSL에 SNI가 없더라도 항상 server_hostname을 전달할 수 있습니다.

버전 3.6에서 변경: *session* 인자가 추가되었습니다.

버전 3.7에서 변경: 이 메서드는 하드 코드 된 *SSLSocket* 대신 *SSLContext.sslsocket_class* 인스턴스를 반환합니다.

`SSLContext.`**`sslsocket_class`**

The return type of *SSLContext.wrap_socket()*, defaults to *SSLSocket*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLSocket*.

버전 3.7에 추가.

`SSLContext.`**`wrap_bio`**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of *SSLContext.sslobject_class* (default *SSLObject*). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

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: 이 메서드는 하드 코드 된 *SSLObject* 대신 *SSLContext.sslobject_class* 인스턴스를 반환합니다.

`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.

버전 3.7에 추가.

`SSLContext.`**`session_stats`**`()`

이 컨텍스트에 의해 생성되거나 관리된 SSL 세션에 대한 통계를 가져옵니다. 각 정보 조각의 이름을 숫자 값에 매핑하는 딕셔너리가 반환됩니다. 예를 들어, 다음은 컨텍스트가 생성된 이후 세션 캐시의 총 적중 횟수 및 누락 횟수입니다:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

`SSLContext.`**`check_hostname`**

Whether to match the peer cert's hostname with *match_hostname()* in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname. Enabling hostname checking automatically sets *verify_mode* from *CERT_NONE* to *CERT_REQUIRED*. It cannot be set back to *CERT_NONE* as long as hostname checking is enabled. The *PROTOCOL_TLS_CLIENT* protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

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))
```

버전 3.4에 추가.

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.

참고: 이 기능을 사용하려면 OpenSSL 0.9.8f 이상이 필요합니다.

SSLContext.**maximum_version**

지원되는 가장 높은 TLS 버전을 나타내는 *TLSVersion* 열거형 멤버. 기본값은 *TLSVersion.MAXIMUM_SUPPORTED*입니다. 어트리뷰트는 *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT* 및 *PROTOCOL_TLS_SERVER* 이외의 프로토콜에 대해 읽기 전용입니다.

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.

참고: ssl 모듈을 OpenSSL 1.1.0g 이상으로 컴파일하지 않으면 이 어트리뷰트를 사용할 수 없습니다.

버전 3.7에 추가.

SSLContext.**minimum_version**

가장 낮은 지원 버전 또는 *TLSVersion.MINIMUM_SUPPORTED*이라는 것만 제외하면 *SSLContext.maximum_version*과 같습니다.

참고: ssl 모듈을 OpenSSL 1.1.0g 이상으로 컴파일하지 않으면 이 어트리뷰트를 사용할 수 없습니다.

버전 3.7에 추가.

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.

참고: 0.9.8m보다 오래된 OpenSSL 버전에서는, 옵션을 지우지는 못하고 설정만 할 수 있습니다. (해당 비트를 재설정하여) 옵션을 지우려고 하면 *ValueError*가 발생합니다.

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>
```

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.

참고: OpenSSL 1.1.1과 TLS 1.3이 활성화될 때만 사용할 수 있습니다. TLS 1.3을 지원 없이는, 프로퍼티 값이 None이며 수정할 수 없습니다.

버전 3.7.1에 추가.

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)

참고: OpenSSL 1.1.0 이상에서만 쓰기 가능합니다.

버전 3.7에 추가.

SSLContext.**verify_flags**
인증서 유효성 검사 연산을 위한 플래그. *VERIFY_CRL_CHECK_LEAF*와 같은 플래그를 함께 OR로 연결하여 설정할 수 있습니다. 기본적으로 OpenSSL은 인증서 해지 목록(CRL)을 요구하지도 확인하지도 않습니다. openssl 버전 0.9.8+ 에서만 사용할 수 있습니다.

버전 3.4에 추가.

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>
```

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

종종 개인 키는 인증서와 같은 파일에 저장됩니다; 이럴 때, *SSLContext.load_cert_chain()*과 *wrap_socket()*에 대한 certfile 매개 변수만 전달하면 됩니다. 개인 키가 인증서와 함께 저장되면, 인증서 체인의 첫 번째 인증서보다 먼저 와야 합니다.:

```
-----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)

The *PROTOCOL_TLS_CLIENT* protocol configures the context for cert validation and hostname verification. *verify_mode* is set to *CERT_REQUIRED* and *check_hostname* is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, *CERT_REQUIRED* and *check_hostname* validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> 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'),),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
           (('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'),
                    ('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.mydomain.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 `connstream`에서 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
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

더 보기:

asyncio 모듈은 비 블로킹 *SSL* 소켓을 지원하며 더 고수준의 API를 제공합니다. *selectors* 모듈을 사용하여 이벤트를 폴링 하고 *SSLWantWriteError*, *SSLWantReadError* 및 *BlockingIOError* 예외를 처리합니다. SSL 핸드 셰이크도 비동기적으로 실행됩니다.

19.3.7 메모리 BIO 지원

버전 3.5에 추가.

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. 이렇게해서 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()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *unwrap()*
- *get_channel_binding()*

*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.
- *wrap_socket()* 에 있는 것과 같은 모듈 수준의 `wrap_bio()` 호출이 없습니다. *SSLObject*는 항상 *SSLContext*를 통해 만들어집니다.

버전 3.7에서 변경: *SSLObject* 인스턴스는 *wrap_bio()*로 만들어야 합니다. 이전 버전에서는, 직접 인스턴스를 만들 수 있었습니다. 이것은 문서로 만들어지거나 공식적으로 지원된 적이 없습니다.

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

버전 3.6에 추가.

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

SSLContext 생성자를 직접 호출할 때, *CERT_NONE*이 기본값입니다. 이것은 다른 피어를 인증하지 않기 때문에, 특히 대부분 대화를 나누려는 서버의 신뢰성을 보장하고 싶어 하는 클라이언트 모드에서는 안전하지 않을 수 있습니다. 따라서 클라이언트 모드에서는, *CERT_REQUIRED*를 사용하는 것이 좋습니다. 그러나 그 자체로는 충분하지 않습니다; *SSLSocket.getpeercert()*를 호출하여 얻을 수 있는 서버 인증서가 원하는 서비스와 일치하는지 확인해야 합니다. 많은 프로토콜과 응용 프로그램에서 서비스는 호스트 이름으로 식별할 수 있습니다; 이럴 때, *match_hostname()* 함수를 사용할 수 있습니다. 이 공통 검사는 *SSLContext.check_hostname*이 활성화되면 자동으로 수행됩니다.

버전 3.7에서 변경: 이제 호스트 이름 일치가 OpenSSL에 의해 수행됩니다. 파이썬은 더는 *match_hostname()*을 사용하지 않습니다.

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.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

위에 만들어진 SSL 컨텍스트는 서버로 TLSv1.2 이상(시스템이 지원한다면)의 연결만 허용합니다. *PROTOCOL_TLS_CLIENT*는 기본적으로 인증서 유효성 검사와 호스트 이름 검사를 의미합니다. 인증서를 컨텍스트에 로드 해야 합니다.

사이퍼 선택

고급 보안 요구 사항이 있으면, *SSLContext.set_ciphers()* 메서드를 통해 SSL 세션을 협상할 때 활성화되는 사이퍼의 미세 조정이 가능합니다. 파이썬 3.2.3부터, ssl 모듈은 기본적으로 특정 약한 사이퍼를 비활성화하지만, 사이퍼 선택을 추가로 제한하려고 할 수 있습니다. 사이퍼 목록 형식에 대한 OpenSSL의 설명서를 읽으십시오. 주어진 사이퍼 목록에 의해 활성화된 사이퍼를 확인하려면, *SSLContext.get_ciphers()* 나 시스템에서 `openssl ciphers` 명령을 사용하십시오.

다중 프로세싱

다중 프로세스 응용 프로그램(예를 들어, : *multiprocessing* 이나 *concurrent.futures* 모듈을 사용하는)의 일부로 이 모듈을 사용하면, OpenSSL의 내부 난수 생성기가 포크 된 프로세스를 제대로 처리하지 못합니다. 응용 프로그램이 *os.fork()*와 함께 SSL 기능을 사용하면, 부모 프로세스의 PRNG 상태를 변경해야 합니다. *RAND_add()*, *RAND_bytes()* 또는 *RAND_pseudo_bytes()*의 성공적인 호출이면 충분합니다.

19.3.10 TLS 1.3

버전 3.7에 추가.

파이썬은 OpenSSL 1.1.1을 사용해서 TLS 1.3에 대한 잠정적이고 실험적인 지원을 제공합니다. 새 프로토콜은 이전 버전의 TLS/SSL과 약간 다르게 동작합니다. 몇몇 새로운 TLS 1.3 기능은 아직 제공되지 않습니다.

- 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.

19.3.11 LibreSSL 지원

LibreSSL은 OpenSSL 1.0.1 포크입니다. ssl 모듈은 LibreSSL을 제한적으로 지원합니다. ssl 모듈이 LibreSSL로 컴파일될 때 일부 기능을 사용할 수 없습니다.

- LibreSSL >= 2.6.1은 더는 NPN을 지원하지 않습니다. *SSLContext.set_npn_protocols()*와 *SSLSocket.selected_npn_protocol()* 메서드는 사용할 수 없습니다.
- *SSLContext.set_default_verify_paths()*는 비록 *get_default_verify_paths()*가 여전히 보고하지만, 환경 변수 `SSL_CERT_FILE`와 `SSL_CERT_PATH`를 무시합니다.

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 (DTL IETF

모질라의 서버 측 TLS 추천 Mozilla

19.4 select — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

참고: The *selectors* module allows high-level and efficient I/O multiplexing, built upon the *select* module primitives. Users are encouraged to use the *selectors* module instead, unless they want precise control over the OS-level primitives used.

The module defines the following:

exception `select.error`
A deprecated alias of *OSError*.

버전 3.3에서 변경: Following **PEP 3151**, this class was made an alias of *OSError*.

`select.`**devpoll**`()`
(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object; see section */dev/poll Polling Objects* below for the methods supported by devpoll objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, `devpoll()` will fail. If your program increases this value, `devpoll()` may return an incomplete list of active file descriptors.

The new file descriptor is *non-inheritable*.

버전 3.3에 추가.

버전 3.4에서 변경: The new file descriptor is now non-inheritable.

`select.`**epoll**`(`*sizehint=-1*, *flags=0*`)`
(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

sizehint informs epoll about the expected number of events to be registered. It must be positive, or *-1* to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the *Edge and Level Trigger Polling (epoll) Objects* section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol: when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is *non-inheritable*.

버전 3.3에서 변경: Added the *flags* parameter.

버전 3.4에서 변경: Support for the `with` statement was added. The new file descriptor is now non-inheritable.

버전 3.4부터 폐지: The *flags* parameter. `select.EPOLL_CLOEXEC` is used by default now. Use *os.set_inheritable()* to make the file descriptor inheritable.

select.**poll**()
(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section *Polling Objects* below for the methods supported by polling objects.

select.**kqueue**()
(Only supported on BSD.) Returns a kernel queue object; see section *Kqueue Objects* below for the methods supported by kqueue objects.

The new file descriptor is *non-inheritable*.

버전 3.4에서 변경: The new file descriptor is now non-inheritable.

select.**kevent**(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)
(Only supported on BSD.) Returns a kernel event object; see section *Kevent Objects* below for the methods supported by kevent objects.

select.**select**(*rlist*, *wlist*, *xlist*[, *timeout*])
This is a straightforward interface to the Unix `select()` system call. The first three arguments are iterables of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named *fileno()* returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty iterables are allowed, but acceptance of three empty iterables is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the iterables are Python *file objects* (e.g. `sys.stdin`, or objects returned by *open()* or *os.popen()*), socket objects returned by *socket.socket()*. You may also define a *wrapper* class yourself, as long as it has an appropriate *fileno()* method (that really returns a file descriptor, not just a random integer).

참고: File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

버전 3.5에서 변경: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

select.**PIPE_BUF**
The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by *select()*, *poll()* or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512.

Availability: Unix

버전 3.2에 추가.

19.4.1 /dev/poll Polling Objects

Solaris and derivatives have /dev/poll. While select() is O(highest file descriptor) and poll() is O(number of file descriptors), /dev/poll is O(active file descriptors).

/dev/poll behaviour is very close to the standard poll() object.

devpoll.**close**()
Close the file descriptor of the polling object.

버전 3.4에 추가.

devpoll.**closed**
True if the polling object is closed.

버전 3.4에 추가.

devpoll.**fileno**()
Return the file descriptor number of the polling object.

버전 3.4에 추가.

devpoll.**register**(*fd*[, *eventmask*])
Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with poll() object. The default value is a combination of the constants POLLIN, POLLPRI, and POLLOUT.

> **경고:** Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with poll().

devpoll.**modify**(*fd*[, *eventmask*])
This method does an *unregister()* followed by a *register()*. It is (a bit) more efficient that doing the same explicitly.

devpoll.**unregister**(*fd*)
Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

devpoll.**poll**([*timeout*])
Polls the set of registered file descriptors, and returns a possibly-empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or *None*, the call will block until there is an event for this poll object.

버전 3.5에서 변경: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.2 Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

Constant	Meaning
EPOLLIN	Available for read
EPOLLOUT	Available for write
EPOLLPRI	Urgent data for read
EPOLLERR	Error condition happened on the assoc. fd
EPOLLHUP	Hang up happened on the assoc. fd
EPOLLET	Set Edge Trigger behavior, the default is Level Trigger behavior
EPOLLONESHOT	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
EPOLLEXCLUSIVE	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
EPOLLRDHUP	Stream socket peer closed connection or shut down writing half of connection.
EPOLLRDNORM	Equivalent to EPOLLIN
EPOLLRDBAND	Priority data band can be read.
EPOLLWRNORM	Equivalent to EPOLLOUT
EPOLLWRBAND	Priority data may be written.
EPOLLMSG	Ignored.

버전 3.6에 추가: EPOLLEXCLUSIVE was added. It's only supported by Linux Kernel 4.5 or later.

`epoll.`**`close`**`()`
Close the control file descriptor of the epoll object.

`epoll.`**`closed`**
`True` if the epoll object is closed.

`epoll.`**`fileno`**`()`
Return the file descriptor number of the control fd.

`epoll.`**`fromfd`**`(`*fd*`)`
Create an epoll object from a given file descriptor.

`epoll.`**`register`**`(`*fd*[, *eventmask*]`)`
Register a fd descriptor with the epoll object.

`epoll.`**`modify`**`(`*fd*, *eventmask*`)`
Modify a registered file descriptor.

`epoll.`**`unregister`**`(`*fd*`)`
Remove a registered file descriptor from the epoll object.

`epoll.`**`poll`**`(`*timeout=-1*, *maxevents=-1*`)`
Wait for events. timeout in seconds (float)

버전 3.5에서 변경: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.3 Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is O(highest file descriptor), while `poll()` is O(number of file descriptors).

`poll.`**`register`**(*fd*[, *eventmask*])

Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constant	Meaning
`POLLIN`	There is data to read
`POLLPRI`	There is urgent data to read
`POLLOUT`	Ready for output: writing will not block
`POLLERR`	Error condition of some sort
`POLLHUP`	Hung up
`POLLRDHUP`	Stream socket peer closed connection, or shut down writing half of connection
`POLLNVAL`	Invalid request: descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

`poll.`**`modify`**(*fd*, *eventmask*)

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an *OSError* exception with errno `ENOENT` to be raised.

`poll.`**`unregister`**(*fd*)

Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a *KeyError* exception to be raised.

`poll.`**`poll`**([*timeout*])

Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or *None*, the call will block until there is an event for this poll object.

버전 3.5에서 변경: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.4 Kqueue Objects

kqueue.**close**()
Close the control file descriptor of the kqueue object.

kqueue.**closed**
`True` if the kqueue object is closed.

kqueue.**fileno**()
Return the file descriptor number of the control fd.

kqueue.**fromfd**(*fd*)
Create a kqueue object from a given file descriptor.

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist
Low level interface to kevent

- changelist must be an iterable of kevent objects or `None`
- max_events must be 0 or a positive integer
- timeout in seconds (floats possible); the default is `None`, to wait forever

버전 3.5에서 변경: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.5 Kevent Objects

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**
Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a *fileno()* method. kevent stores the integer internally.

kevent.**filter**
Name of the kernel filter.

Constant	Meaning
KQ_FILTER_READ	Takes a descriptor and returns whenever there is data available to read
KQ_FILTER_WRITE	Takes a descriptor and returns whenever there is data available to write
KQ_FILTER_AIO	AIO requests
KQ_FILTER_VNODE	Returns when one or more of the requested events watched in *fflag* occurs
KQ_FILTER_PROC	Watch for events on a process id
KQ_FILTER_NETDEV	Watch for events on a network device [not available on Mac OS X]
KQ_FILTER_SIGNAL	Returns whenever the watched signal is delivered to the process
KQ_FILTER_TIMER	Establishes an arbitrary timer

kevent.**flags**
Filter action.

Constant	Meaning
KQ_EV_ADD	Adds or modifies an event
KQ_EV_DELETE	Removes an event from the queue
KQ_EV_ENABLE	Permitscontrol() to returns the event
KQ_EV_DISABLE	Disablesevent
KQ_EV_ONESHOT	Removes event after first occurrence
KQ_EV_CLEAR	Reset the state after an event is retrieved
KQ_EV_SYSFLAGS	internal event
KQ_EV_FLAG1	internal event
KQ_EV_EOF	Filter specific EOF condition
KQ_EV_ERROR	See return values

kevent.**fflags**
Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags:

Constant	Meaning
KQ_NOTE_LOWAT	low water mark of a socket buffer

KQ_FILTER_VNODE filter flags:

Constant	Meaning
KQ_NOTE_DELETE	*unlink()* was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags:

Constant	Meaning
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called *fork()*
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across *fork()*
KQ_NOTE_CHILD	returned on the child process for *NOTE_TRACK*
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on Mac OS X):

Constant	Meaning
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	link state is invalid

kevent.**data**
Filter specific data.

kevent.**udata**
User defined value.

19.5 selectors — 고수준 I/O 다중화

버전 3.4에 추가.

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.

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
EVENT_READ	읽기 가능
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.

버전 3.5에 추가.

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()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    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)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

19.6 asyncore — Asynchronous socket handler

Source code: Lib/asyncore.py

버전 3.6부터 폐지: Please use *asyncio* instead.

참고: This module exists for backwards compatibility only. For new code we recommend using *asyncio*.

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do "more than one thing at a time." Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the `select()` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the "background." Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The *asyncore* module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For "conversational" applications and protocols the companion *asynchat* module is invaluable.

The basic idea behind both modules is to create one or more network *channels*, instances of class *asyncore.dispatcher* and *asynchat.async_chat*. Creating the channels adds them to a global map, used by the *loop()* function if you do not provide it with your own *map*.

Once the initial channel(s) is(are) created, calling the *loop()* function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

`asyncore.`**`loop`**`(`[*timeout*[, *use_poll*[, *map*[, *count*]]]]`)`

Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The *count* parameter defaults to `None`, resulting in the loop terminating only when all channels have been closed. The *timeout* argument sets the timeout parameter for the appropriate *select()* or *poll()* call, measured in seconds; the default is 30 seconds. The *use_poll* parameter, if true, indicates that *poll()* should be used in preference to *select()* (the default is `False`).

The *map* parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If *map* is omitted, a global map is used. Channels (instances of *asyncore.dispatcher*, *asynchat.async_chat* and subclasses thereof) can freely be mixed in the map.

class `asyncore.`**`dispatcher`**

The *dispatcher* class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

Event	Description
`handle_connect()`	Implied by the first read or write event
`handle_close()`	Implied by a read event with no data available
`handle_accepted()`	Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's *readable()* and *writable()* methods are used to determine whether the channel's socket should be added to the list of channels `select()`ed or `poll()`ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

`handle_read``()`

Called when the asynchronous loop detects that a `read()` call on the channel's socket will succeed.

`handle_write``()`

Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()
Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

handle_connect()
Called when the active opener's socket actually makes a connection. Might send a "welcome" banner, or initiate a protocol negotiation with the remote endpoint, for example.

handle_close()
Called when the socket is closed.

handle_error()
Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.

handle_accept()
Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a *connect()* call for the local endpoint. Deprecated in version 3.2; use *handle_accepted()* instead.

버전 3.2부터 폐지.

handle_accepted(*sock*, *addr*)
Called on listening channels (passive openers) when a connection has been established with a new remote endpoint that has issued a *connect()* call for the local endpoint. *sock* is a *new* socket object usable to send and receive data on the connection, and *addr* is the address bound to the socket on the other end of the connection.

버전 3.2에 추가.

readable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

writable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

create_socket(*family=socket.AF_INET*, *type=socket.SOCK_STREAM*)
This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the *socket* documentation for information on creating sockets.

버전 3.3에서 변경: *family* and *type* arguments can be omitted.

connect(*address*)
As with the normal socket object, *address* is a tuple with the first element the host to connect to, and the second the port number.

send(*data*)
Send *data* to the remote end-point of the socket.

recv(*buffer_size*)
Read at most *buffer_size* bytes from the socket's remote end-point. An empty bytes object implies that the channel has been closed from the other end.

Note that *recv()* may raise *BlockingIOError* , even though *select.select()* or *select.poll()* has reported the socket ready for reading.

listen(*backlog*)
Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

bind(*address*)
Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — refer to the *socket* documentation for more information.) To mark the socket as re-usable (setting the `SO_REUSEADDR` option), call the *dispatcher* object's `set_reuse_addr()` method.

accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either `None` or a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection. When `None` is returned it means the connection didn't take place, in which case the server should just ignore this event and keep listening for further incoming connections.

close()
Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

class asyncore.**dispatcher_with_send**
A *dispatcher* subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use *asynchat.async_chat*.

class asyncore.**file_dispatcher**
A file_dispatcher takes a file descriptor or *file object* along with an optional map argument and wraps it for use with the `poll()` or `loop()` functions. If provided a file object or anything with a `fileno()` method, that method will be called and passed to the *file_wrapper* constructor.

Availability: Unix.

class asyncore.**file_wrapper**
A file_wrapper takes an integer file descriptor and calls *os.dup()* to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the *file_dispatcher* class.

Availability: Unix.

19.6.1 asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the *dispatcher* class to implement its socket handling:

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

19.6.2 asyncore Example basic echo server

Here is a basic echo server that uses the *dispatcher* class to accept connections and dispatches the incoming connections to a handler:

```
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

19.7 asynchat — Asynchronous socket command/response handler

Source code: Lib/asynchat.py

버전 3.6부터 폐지: Please use *asyncio* instead.

참고: This module exists for backwards compatibility only. For new code we recommend using *asyncio*.

This module builds on the *asyncore* infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. *asynchat* defines the abstract class *async_chat* that you subclass, providing implementations of the `collect_incoming_data()` and `found_terminator()` methods. It uses the same asynchronous loop as *asyncore*, and the two types of channel, *asyncore.dispatcher* and *asynchat.async_chat*, can freely be mixed in the channel map. Typically an *asyncore.dispatcher* server channel generates new *asynchat.async_chat* channel objects as it receives incoming connection requests.

class asynchat.**async_chat**

This class is an abstract subclass of *asyncore.dispatcher*. To make practical use of the code you must subclass *async_chat*, providing meaningful *collect_incoming_data()* and *found_terminator()* methods. The *asyncore.dispatcher* methods can be used, although not all make sense in a message/response context.

Like *asyncore.dispatcher*, *async_chat* defines a set of events that are generated by an analysis of socket conditions after a `select()` call. Once the polling loop has been started the *async_chat* object's methods are called by the event-processing framework with no action on the part of the programmer.

Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

ac_in_buffer_size

The asynchronous input buffer size (default `4096`).

ac_out_buffer_size

The asynchronous output buffer size (default `4096`).

Unlike *asyncore.dispatcher*, *async_chat* allows you to define a FIFO queue of *producers*. A producer need have only one method, `more()`, which should return data to be transmitted on the channel. The producer indicates exhaustion (*i.e.* that it contains no more data) by having its `more()` method return the empty bytes object. At this point the *async_chat* object removes the producer from the queue and starts using the next producer, if any. When the producer queue is empty the `handle_write()` method does nothing. You use the channel object's *set_terminator()* method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning *async_chat* subclass your input methods *collect_incoming_data()* and *found_terminator()* must handle the data that the channel receives asynchronously. The methods are described below.

async_chat.**close_when_done**()

Pushes a `None` on to the producer queue. When this producer is popped off the queue it causes the channel to be closed.

async_chat.**collect_incoming_data**(*data*)

Called with *data* holding an arbitrary amount of received data. The default method, which must be overridden, raises a *NotImplementedError* exception.

async_chat.**discard_buffers**()

In emergencies this method will discard any data held in the input and/or output buffers and the producer queue.

async_chat.**found_terminator**()
Called when the incoming data stream matches the termination condition set by *set_terminator()*. The default method, which must be overridden, raises a *NotImplementedError* exception. The buffered input data should be available via an instance attribute.

async_chat.**get_terminator**()
Returns the current terminator for the channel.

async_chat.**push**(*data*)
Pushes data on to the channel's queue to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

async_chat.**push_with_producer**(*producer*)
Takes a producer object and adds it to the producer queue associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer's data by calling its `more()` method and send the data to the remote endpoint.

async_chat.**set_terminator**(*term*)
Sets the terminating condition to be recognized on the channel. `term` may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

term	Description
string	Will call *found_terminator()* when the string is found in the input stream
integer	Will call *found_terminator()* when the indicated number of characters have been received
`None`	The channel continues to collect data forever

Note that any data following the terminator will be available for reading by the channel after *found_terminator()* is called.

19.7.1 asynchat Example

The following partial example shows how HTTP requests can be read with *async_chat*. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

19.8 `signal` — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.

19.8.1 General rules

The *signal.signal()* function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: *SIGPIPE* is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and *SIGINT* is translated into a *KeyboardInterrupt* exception if the parent process has not changed it.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for *SIGCHLD*, which follows the underlying implementation.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the *virtual machine* to execute the corresponding Python signal handler at a later point(for example at the next *bytecode* instruction). This has consequences:

- It makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV` that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the `faulthandler` module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

Signals and threads

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can't be used as a means of inter-thread communication. You can use the synchronization primitives from the `threading` module instead.

Besides, only the main thread is allowed to set a new signal handler.

19.8.2 Module contents

버전 3.5에서 변경: signal (SIG*), handler (`SIG_DFL`, `SIG_IGN`) and sigmask (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into `enums`. `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable `enums`.

The variables defined in the `signal` module are:

`signal.`**`SIG_DFL`**
: This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for `SIGQUIT` is to dump core and exit, while the default action for `SIGCHLD` is to simply ignore it.

`signal.`**`SIG_IGN`**
: This is another standard signal handler, which will simply ignore the given signal.

`signal.`**`SIGABRT`**
: Abort signal from `abort(3)`.

`signal.`**`SIGALRM`**
: Timer signal from `alarm(2)`.

  Availability: Unix.

`signal.`**`SIGBREAK`**
: Interrupt from keyboard (CTRL + BREAK).

  Availability: Windows.

`signal.`**`SIGBUS`**
: Bus error (bad memory access).

  Availability: Unix.

signal.**SIGCHLD**
Child process stopped or terminated.

Availability: Windows.

signal.**SIGCLD**
Alias to *SIGCHLD*.

signal.**SIGCONT**
Continue the process if it is currently stopped

Availability: Unix.

signal.**SIGFPE**
Floating-point exception. For example, division by zero.

더 보기:

ZeroDivisionError is raised when the second argument of a division or modulo operation is zero.

signal.**SIGHUP**
Hangup detected on controlling terminal or death of controlling process.

Availability: Unix.

signal.**SIGILL**
Illegal instruction.

signal.**SIGINT**
Interrupt from keyboard (CTRL + C).

Default action is to raise *KeyboardInterrupt*.

signal.**SIGKILL**
Kill signal.

It cannot be caught, blocked, or ignored.

Availability: Unix.

signal.**SIGPIPE**
Broken pipe: write to pipe with no readers.

Default action is to ignore the signal.

Availability: Unix.

signal.**SIGSEGV**
Segmentation fault: invalid memory reference.

signal.**SIGTERM**
Termination signal.

signal.**SIGUSR1**
User-defined signal 1.

Availability: Unix.

signal.**SIGUSR2**
User-defined signal 2.

Availability: Unix.

signal.**SIGWINCH**
Window resize signal.

Availability: Unix.

SIG*
All the signal numbers are defined symbolically. For example, the hangup signal is defined as *signal.SIGHUP*; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for 'signal()' lists the existing signals (on some systems this is *signal(2)*, on others the list is in *signal(7)*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.**CTRL_C_EVENT**
The signal corresponding to the Ctrl+C keystroke event. This signal can only be used with *os.kill()*.

Availability: Windows.

버전 3.2에 추가.

signal.**CTRL_BREAK_EVENT**
The signal corresponding to the Ctrl+Break keystroke event. This signal can only be used with *os.kill()*.

Availability: Windows.

버전 3.2에 추가.

signal.**NSIG**
One more than the number of the highest signal number.

signal.**ITIMER_REAL**
Decrements interval timer in real time, and delivers *SIGALRM* upon expiration.

signal.**ITIMER_VIRTUAL**
Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.**ITIMER_PROF**
Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

signal.**SIG_BLOCK**
A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be blocked.

버전 3.3에 추가.

signal.**SIG_UNBLOCK**
A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be unblocked.

버전 3.3에 추가.

signal.**SIG_SETMASK**
A possible value for the *how* parameter to *pthread_sigmask()* indicating that the signal mask is to be replaced.

버전 3.3에 추가.

The *signal* module defines one exception:

exception signal.**ItimerError**
Raised to signal an error from the underlying *setitimer()* or *getitimer()* implementation. Expect this error if an invalid interval timer or a negative time is passed to *setitimer()*. This error is a subtype of *OSError*.

버전 3.3에 추가: This error used to be a subtype of *IOError*, which is now an alias of *OSError*.

The *signal* module defines the following functions:

signal.**alarm**(*time*)
If *time* is non-zero, this function requests that a *SIGALRM* signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then

the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled.

Availability: Unix. See the man page *alarm(2)* for further information.

signal.**getsignal**(*signalnum*)
Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values *signal.SIG_IGN*, *signal.SIG_DFL* or *None*. Here, *signal.SIG_IGN* means that the signal was previously ignored, *signal.SIG_DFL* means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

signal.**pause**()
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing.

Availability: Unix. See the man page *signal(2)* for further information.

See also *sigwait()*, *sigwaitinfo()*, *sigtimedwait()* and *sigpending()*.

signal.**pthread_kill**(*thread_id*, *signalnum*)
Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with *InterruptedError*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

Availability: Unix. See the man page *pthread_kill(3)* for further information.

See also *os.kill()*.

버전 3.3에 추가.

signal.**pthread_sigmask**(*how*, *mask*)
Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- *SIG_BLOCK*: The set of blocked signals is the union of the current set and the *mask* argument.
- *SIG_UNBLOCK*: The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- *SIG_SETMASK*: The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {*signal.SIGINT*, *signal.SIGTERM*}). Use `range(1, signal.NSIG)` for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

SIGKILL and SIGSTOP cannot be blocked.

Availability: Unix. See the man page *sigprocmask(3)* and *pthread_sigmask(3)* for further information.

See also *pause()*, *sigpending()* and *sigwait()*.

버전 3.3에 추가.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)
Sets given interval timer (one of *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* or *signal.ITIMER_PROF*) specified by *which* to fire after *seconds* (float is accepted, different from *alarm()*) and after that every *interval* seconds (if *interval* is non-zero). The interval timer specified by *which* can be cleared by setting *seconds* to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; *signal.ITIMER_REAL* will deliver *SIGALRM*, *signal.ITIMER_VIRTUAL* sends SIGVTALRM, and *signal.ITIMER_PROF* will deliver SIGPROF.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an *ItimerError*.

Availability: Unix.

signal.**getitimer**(*which*)
Returns current value of a given interval timer specified by *which*.

Availability: Unix.

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)
Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

버전 3.5에서 변경: On Windows, the function now also supports socket handles.

버전 3.7에서 변경: Added `warn_on_full_buffer` parameter.

signal.**siginterrupt**(*signalnum*, *flag*)
Change system call restart behaviour: if *flag* is *False*, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing.

Availability: Unix. See the man page *siginterrupt(3)* for further information.

Note that installing a signal handler with *signal()* will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true *flag* value for the given signal.

signal.**signal**(*signalnum*, *handler*)
Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two

arguments (see below), or one of the special values *signal.SIG_IGN* or *signal.SIG_DFL*. The previous signal handler will be returned (see the description of *getsignal()* above). (See the Unix man page *signal(2)* for further information.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the *inspect* module).

On Windows, *signal()* can only be called with *SIGABRT*, *SIGFPE*, *SIGILL*, *SIGINT*, *SIGSEGV*, *SIGTERM*, or *SIGBREAK*. A *ValueError* will be raised in any other case. Note that not all systems define the same set of signal names; an *AttributeError* will be raised if a signal name is not defined as `SIG*` module level constant.

`signal.`**`sigpending`**`()`

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix. See the man page *sigpending(2)* for further information.

See also *pause()*, *pthread_sigmask()* and *sigwait()*.

버전 3.3에 추가.

`signal.`**`sigwait`**`(`*sigset*`)`

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix. See the man page *sigwait(3)* for further information.

See also *pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* and *sigtimedwait()*.

버전 3.3에 추가.

`signal.`**`sigwaitinfo`**`(`*sigset*`)`

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an *InterruptedError* if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability: Unix. See the man page *sigwaitinfo(2)* for further information.

See also *pause()*, *sigwait()* and *sigtimedwait()*.

버전 3.3에 추가.

버전 3.5에서 변경: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

`signal.`**`sigtimedwait`**`(`*sigset*, *timeout*`)`

Like *sigwaitinfo()*, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as `0`, a poll is performed. Returns *None* if a timeout occurs.

Availability: Unix. See the man page *sigtimedwait(2)* for further information.

See also *pause()*, *sigwait()* and *sigwaitinfo()*.

버전 3.3에 추가.

버전 3.5에서 변경: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

19.8.3 Example

Here is a minimal example program. It uses the *alarm()* function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the *os.open()* to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

19.8.4 Note on SIGPIPE

Piping output of your program to tools like *head(1)* will cause a *SIGPIPE* signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set *SIGPIPE*'s disposition to *SIG_DFL* in order to avoid *BrokenPipeError*. Doing that would cause your program to exit unexpectedly also whenever any socket connection is interrupted while your program is still writing to it.

19.9 mmap — 메모리 맵 파일 지원

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_DEFAULT를 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*])

(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 None이 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. 태그 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 ALLOCATIONGRANULARITY의 배수여야 합니다.

class mmap.**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])

(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_PRIVATE는 비공개 쓸 때 복사 (copy-on-write) 매핑을 생성하므로, mmap 객체의 내용에 대한 변경 사항은 이 프로세스에만 적용되고, MAP_SHARED는 파일의 같은 영역을 매핑하는 다른 모든 프로세스와 공유되는 매핑을 만듭니다. 기본값은 MAP_SHARED입니다.

*prot*가 지정되면 원하는 메모리 보호를 제공합니다; 가장 유용한 두 값은 페이지를 읽거나 쓰도록 지정할 수 있는 PROT_READ와 PROT_WRITE입니다. *prot*의 기본값은 PROT_READ | PROT_WRITE입니다.

*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 PAGESIZE와 같은 ALLOCATIONGRANULARITY의 배수여야 합니다.

생성된 메모리 매핑의 유효성을 보장하기 위해, 기술자 *fileno*로 지정된 파일은 맥 OS X 및 OpenVMS의 물리적 저장 장치와 내부적으로 자동 동기화됩니다.

이 예제는 *mmap*을 사용하는 간단한 방법을 보여줍니다:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    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!")
```

버전 3.2에 추가: 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:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

메모리 맵 파일 객체는 다음 메서드를 지원합니다:

close()
mmap를 닫습니다. 이후에 객체의 다른 메서드를 호출하면 ValueError 예외가 발생합니다. 열려있는 파일을 닫지 않습니다.

closed
파일이 닫혔으면 `True`입니다.

버전 3.2에 추가.

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([*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.

(윈도우 버전) 반환된 0이 아닌 값은 성공을 나타냅니다; 0은 실패를 나타냅니다.

(유닉스 버전) 성공을 나타내기 위해 0 값이 반환됩니다. 호출이 실패하면 예외가 발생합니다.

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.

resize(*newsize*)
맵과 하부 파일(있다면)의 크기를 조정합니다. mmap이 `ACCESS_READ` 나 `ACCESS_COPY`로 만들어졌을 때, 맵의 크기를 조정하면 *TypeError* 예외가 발생합니다.

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.

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.

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*와 *nntplib* 같은 모듈의 기능입니다. *email* 패키지는 **RFC 5233**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 *parser*와 *generator*입니다. 구문 분석기(parser)는 직렬화된 전자 메일 메시지(바이트 스트림)를 가져와 *EmailMessage* 객체의 트리로 변환합니다. 생성기(generator)는 *EmailMessage*를 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 *parser*에는 그것의 동작을 제어하는 연관된 *policy* 객체가 있습니다. 일반적으로 응용 프로그램은 *EmailMessage*가 만들어질 때 정책을 지정하기만 하면 되는데, *EmailMessage*를 직접 인스턴스로 만들어서 새 전자 메일을 만들거나, *parser*를 사용하여 입력 스트림을 구문 분석할 때입니다. 그러나 메시지가 *generator*를 사용하여 직렬화될

때 정책을 변경할 수 있습니다. 이것은, 예를 들어, 범용 전자 메일 메시지를 디스크에서 구분 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`: Representing an email message

Source code: Lib/email/message.py

버전 3.6에 추가:[1]

The central class in the `email` package is the `EmailMessage` class, imported from the `email.message` module. It is the base class for the `email` object model. `EmailMessage` provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of *headers* and a *payload* (which is also referred to as the *content*). Headers are **RFC 5322** or **RFC 6532** style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as `multipart/*` or `message/rfc822`.

The conceptual model provided by an `EmailMessage` object is that of an ordered dictionary of headers coupled with a *payload* that represents the **RFC 5322** body of the message, which might be a list of sub-`EmailMessage` objects.

[1] Originally added in 3.4 as a *provisional module*. Docs for legacy message class moved to *email.message.Message: Representing an email message using the compat32 API*.

In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for generating a serialized version of the message, and for recursively walking over the object tree.

The *EmailMessage* dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. Unlike a real dict, there is an ordering to the keys, and there can be duplicate keys. Additional methods are provided for working with headers that have duplicate keys.

The *payload* is either a string or bytes object, in the case of simple message objects, or a list of *EmailMessage* objects, for MIME container documents such as *multipart/** and *message/rfc822* message objects.

class email.message.**EmailMessage**(*policy=default*)

If *policy* is specified use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *default* policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated `\r\n`, it uses the Python standard `\n` line endings). For more information see the *policy* documentation.

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base *Message* class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the `max_line_length` of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *Generator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.Generator* for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as "7 bit clean" when *utf8* is `False`, which is the default.

버전 3.6에서 변경: the default behavior when *maxheaderlen* is not specified was changed from defaulting to 0 to defaulting to the value of *max_line_length* from the policy.

__str__()

Equivalent to `as_string(policy=self.policy.clone(utf8=True))`. Allows `str(msg)` to produce a string containing the serialized message in a readable format.

버전 3.4에서 변경: the method was changed to use `utf8=True`, thus producing an **RFC 6531**-like message representation, instead of being a direct alias for *as_string()*.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *BytesGenerator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.BytesGenerator* for a more flexible API for serializing messages.

__bytes__()

Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

is_multipart()
Return True if the message's payload is a list of sub-*EmailMessage* objects, otherwise return False. When *is_multipart()* returns False, the payload should be a string object (which might be a CTE encoded binary payload). Note that *is_multipart()* returning True does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the True. For example, is_multipart will return True when the *EmailMessage* is of type message/rfc822.

set_unixfrom(*unixfrom*)
Set the message's envelope header to *unixfrom*, which should be a string. (See *mboxMessage* for a brief description of this header.)

get_unixfrom()
Return the message's envelope header. Defaults to None if the envelope header was never set.

The following methods implement the mapping-like interface for accessing the message's headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in an *EmailMessage* object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()
Return the total number of headers, including duplicates.

__contains__(*name*)
Return True if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the in operator. For example:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
Return the value of the named header field. *name* does not include the colon field separator. If the header is missing, None is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant headers named *name*.

Using the standard (non-compat32) policies, the returned value is an instance of a subclass of *email.headerregistry.BaseHeader*.

__setitem__(*name*, *val*)
Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing headers.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the policy defines certain headers to be unique (as the standard policies do), this method may raise a *ValueError* when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__(*name*)
Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)
Return the value of the named header field. This is identical to *__getitem__()* except that optional *failobj* is returned if the named header is missing (*failobj* defaults to `None`).

Here are some additional useful header related methods:

get_all(*name*, *failobj=None*)
Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)
Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here is an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example of the extended interface with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)
Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case of the original header. If no matching header is found, raise a *KeyError*.

get_content_type()
Return the message's content type, coerced to lower case of the form *maintype/subtype*. If there is no *Content-Type* header in the message return the value returned by *get_default_type()*. If the *Content-Type* header is invalid, return `text/plain`.

(According to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value. **RFC 2045** defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.)

get_content_maintype()
Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()
Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()
Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)
Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header, so it only affects the return value of the `get_content_type` methods when no *Content-Type* header is present in the message.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is `Content-Type` (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the charset and language may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the `utf8` *charset* and `None` for the *language*.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

Note that existing parameter values of headers may be accessed through the `params` attribute of the header value (for example, `msg['Content-Type'].params['charset']`).

버전 3.4에서 변경: `replace` keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)
Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

get_filename(*failobj=None*)
Return the value of the `filename` parameter of the *Content-Disposition* header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary(*failobj=None*)
Return the value of the `boundary` parameter of the *Content-Type* header of the message, or *failobj* if

either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)
Set the `boundary` parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different from deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers.

get_content_charset(*failobj=None*)
Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

get_charsets(*failobj=None*)
Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. If the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

is_attachment()
Return `True` if there is a *Content-Disposition* header and its (case insensitive) value is `attachment`, `False` otherwise.

버전 3.4.2에서 변경: is_attachment is now a method instead of a property, for consistency with *is_multipart()*.

get_content_disposition()
Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183**.

버전 3.5에 추가.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()
The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> 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
```

`walk` iterates over the subparts of any part where *is_multipart()* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> 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
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

get_body (*preferencelist=('related', 'html', 'plain')*)
Return the MIME part that is the best candidate to be the "body" of the message.

preferencelist must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the `get_body` method is called.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a `multipart/related`, check the `start` parameter and if a part with a matching *Content-ID* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the `multipart/related`.

If a part has a *Content-Disposition* header, only consider the part a candidate match if the value of the header is `inline`.

If none of the candidates matches any of the preferences in *preferencelist*, return `None`.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are `('plain',)`, `('html', 'plain')`, and the default `('related', 'html', 'plain')`. (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a `multipart/related` will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *Content-Type* or whose *Content-Type* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

iter_attachments()
Return an iterator over all of the immediate sub-parts of the message that are not candidate "body" parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *Content-Disposition: attachment*), and return all remaining parts. When applied directly to a `multipart/related`, return an iterator over the all the related parts except the root part (ie: the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn't match the *Content-ID* of any of the parts). When applied directly to a `multipart/alternative` or a non-`multipart`, return an empty iterator.

iter_parts()
Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-multipart. (See also *walk()*.)

get_content(**args*, *content_manager=None*, ***kw*)
Call the *get_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

set_content(**args*, *content_manager=None*, ***kw*)
Call the *set_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

make_related(*boundary=None*)
Convert a non-multipart message into a multipart/related message, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_alternative(*boundary=None*)
Convert a non-multipart or a multipart/related into a multipart/alternative, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed(*boundary=None*)
Convert a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related(**args*, *content_manager=None*, ***kw*)
If the message is a multipart/related, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, call *make_related()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value inline.

add_alternative(**args*, *content_manager=None*, ***kw*)
If the message is a multipart/alternative, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart or multipart/related, call *make_alternative()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

add_attachment(**args*, *content_manager=None*, ***kw*)
If the message is a multipart/mixed, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, multipart/related, or multipart/alternative, call *make_mixed()* and then proceed as above. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value attachment. This method can be used both for explicit attachments (*Content-Disposition: attachment*) and inline attachments (*Content-Disposition: inline*), by passing appropriate options to the content_manager.

clear()
Remove the payload and all of the headers.

clear_content()
Remove the payload and all of the `Content-` headers, leaving all other headers intact and in their original order.

EmailMessage objects have the following instance attributes:

preamble
The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *email.parser* and *email.generator* for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue
The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message. As with the *preamble*, if there is no epilog text this attribute will be `None`.

defects
The *defects* attribute contains a list of all the problems found when parsing this message. See *email.errors* for a detailed description of the possible parsing defects.

class email.message.**MIMEPart**(*policy=default*)
This class represents a subpart of a MIME message. It is identical to *EmailMessage*, except that no *MIME-Version* headers are added when *set_content()* is called, since sub-parts do not need their own *MIME-Version* headers.

20.1.2 email.parser: Parsing email messages

Source code: Lib/email/parser.py

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an *EmailMessage* object, adding headers using the dictionary interface, and adding payload(s) using *set_content()* and related methods, or they can be created by parsing a serialized representation of the email message.

The *email* package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root *EmailMessage* instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its *is_multipart()* method, and the subparts can be accessed via the payload manipulation methods, such as *get_body()*, *iter_parts()*, and *walk()*.

There are actually two parser interfaces available for use, the *Parser* API and the incremental *FeedParser* API. The *Parser* API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. *FeedParser* is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The *FeedParser* can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the *email* package's bundled parser and the *EmailMessage* class is embodied

in the `policy` class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `policy` methods.

FeedParser API

The *BytesFeedParser*, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The *BytesFeedParser* can of course be used to parse an email message fully contained in a *bytes-like object*, string, or file, but the *BytesParser* API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The *BytesFeedParser*'s API is simple; you create an instance, feed it a bunch of bytes until there's no more to feed it, then close the parser to retrieve the root message object. The *BytesFeedParser* is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's *defects* attribute with a list of any problems it found in a message. See the *email.errors* module for the list of defects that it can find.

Here is the API for the *BytesFeedParser*:

class email.parser.**BytesFeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

Create a *BytesFeedParser* instance. Optional *_factory* is a no-argument callable; if not specified use the *message_factory* from the *policy*. Call *_factory* whenever a new message object is needed.

If *policy* is specified use the rules it specifies to update the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides *Message* as the default factory. All other policies provide *EmailMessage* as the default *_factory*. For more information on what else *policy* controls, see the *policy* documentation.

Note: **The policy keyword should always be specified**; The default will change to *email.policy.default* in a future version of Python.

버전 3.2에 추가.

버전 3.3에서 변경: Added the *policy* keyword.

버전 3.6에서 변경: *_factory* defaults to the policy `message_factory`.

feed(*data*)

Feed the parser some more data. *data* should be a *bytes-like object* containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings: carriage return, newline, or carriage return and newline (they can even be mixed).

close()

Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if *feed()* is called after this method has been called.

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: Added the *policy* keyword.

Parser API

The `BytesParser` class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a *bytes-like object* or file. The `email.parser` module also provides `Parser` for parsing strings, and header-only parsers, `BytesHeaderParser` and `HeaderParser`, which can be used if you're only interested in the headers of the message. `BytesHeaderParser` and `HeaderParser` can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

class email.parser.**BytesParser**(*_class=None*, ***, *policy=policy.compat32*)

Create a `BytesParser` instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of `BytesFeedParser`.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

버전 3.3에서 변경: Removed the *strict* argument that was deprecated in 2.4. Added the *policy* keyword.

버전 3.6에서 변경: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the `readline()` and the `read()` methods.

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`).

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

parsebytes(*bytes*, *headersonly=False*)

Similar to the `parse()` method, except it takes a *bytes-like object* instead of a file-like object. Calling this method on a *bytes-like object* is equivalent to wrapping *bytes* in a `BytesIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

버전 3.2에 추가.

class email.parser.**BytesHeaderParser**(*_class=None*, ***, *policy=policy.compat32*)

Exactly like `BytesParser`, except that *headersonly* defaults to `True`.

버전 3.3에 추가.

class email.parser.**Parser**(*_class=None*, ***, *policy=policy.compat32*)

This class is parallel to `BytesParser`, but handles string input.

버전 3.3에서 변경: Removed the *strict* argument. Added the *policy* keyword.

버전 3.6에서 변경: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the text-mode file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the `readline()` and the `read()` methods on file-like objects.

Other than the text mode requirement, this method operates like `BytesParser.parse()`.

parsestr(*text*, *headersonly=False*)

Similar to the `parse()` method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping *text* in a `StringIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

class email.parser.**HeaderParser**(*_class=None*, ***, *policy=policy.compat32*)
Exactly like `Parser`, except that *headersonly* defaults to `True`.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level `email` package namespace.

email.**message_from_bytes**(*s*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure from a *bytes-like object*. This is equivalent to `BytesParser().parsebytes(s)`. Optional *_class* and *policy* are interpreted as with the `BytesParser` class constructor.

버전 3.2에 추가.

버전 3.3에서 변경: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_binary_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure tree from an open binary *file object*. This is equivalent to `BytesParser().parse(fp)`. *_class* and *policy* are interpreted as with the `BytesParser` class constructor.

버전 3.2에 추가.

버전 3.3에서 변경: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_string**(*s*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure from a string. This is equivalent to `Parser().parsestr(s)`. *_class* and *policy* are interpreted as with the `Parser` class constructor.

버전 3.3에서 변경: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure tree from an open *file object*. This is equivalent to `Parser().parse(fp)`. *_class* and *policy* are interpreted as with the `Parser` class constructor.

버전 3.3에서 변경: Removed the *strict* argument. Added the *policy* keyword.

버전 3.6에서 변경: *_class* defaults to the policy `message_factory`.

Here's an example of how you might use `message_from_bytes()` at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Additional notes

Here are some notes on the parsing semantics:

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for `is_multipart()`, and `iter_parts()` will yield an empty list.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for `is_multipart()`, and `iter_parts()` will yield a list of subparts.
- Most messages with a content type of *message/** (such as *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1. Their `is_multipart()` method will return `True`. The single element yielded by `iter_parts()` will be a sub-message object.
- Some non-standards-compliant messages may not be internally consistent about their *multipart*-edness. Such messages may have a *Content-Type* header of type *multipart*, but their `is_multipart()` method may return `False`. If such messages were parsed with the `FeedParser`, they will have an instance of the

`MultipartInvariantViolationDefect` class in their *defects* attribute list. See `email.errors` for details.

20.1.3 `email.generator`: Generating MIME documents

Source code: Lib/email/generator.py

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via `smtplib.SMTP.sendmail()` or the `nntplib` module, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the `email.parser` module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming `policy` is used for both. That is, parsing the serialized byte stream via the `BytesParser` class and then regenerating the serialized byte stream using `BytesGenerator` should produce output identical to the input[1]. (On the other hand, using the generator on an `EmailMessage` constructed by program may result in changes to the `EmailMessage` object as defaults are filled in.)

The `Generator` class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not "8 bit clean".

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a `BytesGenerator` object that will write any message provided to the `flatten()` method, or any surrogateescape encoded text provided to the `write()` method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts binary data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the `mangle_from_` setting of the *policy* (which is `True` for the `compat32` policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see `mailbox` and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the `Message` or `EmailMessage` object passed to `flatten` to control the message generation. See `email.policy` for details on what *policy* controls.

버전 3.2에 추가.

버전 3.3에서 변경: Added the *policy* keyword.

버전 3.6에서 변경: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

[1] This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no `policy` settings calling for automatic adjustments (for example, `refold_source` must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

flatten(*msg*, *unixfrom=False*, *linesep=None*)
Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `BytesGenerator` instance was created.

If the `policy` option `cte_type` is `8bit` (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII `Content-Transfer-Encoding` of any body parts that have them. If `cte_type` is `7bit`, convert the bytes with the high bit set as needed using an ASCII-compatible `Content-Transfer-Encoding`. That is, transform parts with non-ASCII `Content-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) to an ASCII compatible `Content-Transfer-Encoding`, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

clone(*fp*)
Return an independent clone of this `BytesGenerator` instance with the exact same option settings, and *fp* as the new *outfp*.

write(*s*)
Encode *s* using the `ASCII` codec and the `surrogateescape` error handler, and pass it to the *write* method of the *outfp* passed to the `BytesGenerator`'s constructor.

As a convenience, `EmailMessage` provides the methods `as_bytes()` and `bytes(aMessage)` (a.k.a. `__bytes__()`), which simplify the generation of a serialized binary representation of a message object. For more detail, see `email.message`.

Because strings cannot represent binary data, the `Generator` class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible `Content-Transfer_Encoding`. Using the terminology of the email RFCs, you can think of this as `Generator` serializing to an I/O stream that is not "8 bit clean". In other words, most applications will want to be using `BytesGenerator`, and not `Generator`.

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)
Return a `Generator` object that will write any message provided to the `flatten()` method, or any text provided to the `write()` method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts string data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the `mangle_from_` setting of the *policy* (which is `True` for the `compat32` policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see `mailbox` and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the `Message` or `EmailMessage` object passed to `flatten` to control the message generation. See `email.policy` for details on what *policy* controls.

버전 3.3에서 변경: Added the *policy* keyword.

버전 3.6에서 변경: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)
Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `Generator` instance was created.

If the `policy` option `cte_type` is `8bit`, generate the message as if the option were set to `7bit`. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible `Content-Transfer-Encoding`. That is, transform parts with non-ASCII `Content-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) to an ASCII compatible `Content-Transfer-Encoding`, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

버전 3.2에서 변경: Added support for re-encoding `8bit` message bodies, and the *linesep* argument.

clone(*fp*)
Return an independent clone of this `Generator` instance with the exact same options, and *fp* as the new *outfp*.

write(*s*)
Write *s* to the *write* method of the *outfp* passed to the `Generator`'s constructor. This provides just enough file-like API for `Generator` instances to be used in the `print()` function.

As a convenience, `EmailMessage` provides the methods `as_string()` and `str(aMessage)` (a.k.a. `__str__()`), which simplify the generation of a formatted string representation of a message object. For more detail, see `email.message`.

The `email.generator` module also provides a derived class, `DecodedGenerator`, which is like the `Generator` base class, except that non-*text* parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, ***, *policy=None*)
Act like `Generator`, except that for any subpart of the message passed to `Generator.flatten()`, if the subpart is of main type *text*, print the decoded payload of the subpart, and if the main type is not *text*, instead of printing it fill in the string *fmt* using information from the part and print the resulting filled-in string.

To fill in *fmt*, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values:

- `type` – Full MIME type of the non-*text* part
- `maintype` – Main MIME type of the non-*text* part
- `subtype` – Sub-MIME type of the non-*text* part
- `filename` – Filename of the non-*text* part
- `description` – Description associated with the non-*text* part
- `encoding` – Content transfer encoding of the non-*text* part

If *fmt* is `None`, use the following default *fmt*:

"[Non-text (%(type)s) part of message omitted, filename %(filename)s]"

Optional *_mangle_from_* and *maxheaderlen* are as with the `Generator` base class.

20.1.4 email.policy: Policy Objects

버전 3.3에 추가.

Source code: Lib/email/policy.py

The *email* package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary 'body'), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A *Policy* object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. *Policy* instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the *parser* classes and the related convenience functions, and for the *Message* class, this is the *Compat32* policy, via its corresponding pre-defined instance *compat32*. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the *policy* keyword to *EmailMessage* is the *EmailPolicy* policy, via its pre-defined instance *default*.

When a *Message* or *EmailMessage* object is created, it acquires a policy. If the message is created by a *parser*, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a *generator*, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the *policy* keyword for the *email.parser* classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the *parser* module.

The first part of this documentation covers the features of *Policy*, an *abstract base class* that defines the features that are common to all policy objects, including *compat32*. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes *EmailPolicy* and *Compat32*, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

Policy instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new *Policy* instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```
>>> 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'))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling *BytesGenerator* to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the *as_bytes()* method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```
>>> 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*)

This is the *abstract base class* for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the *clone()* method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

max_line_length

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per **RFC 5322**. A value of `0` or *None* indicates that no line wrapping should be done at all.

linesep

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

cte_type

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

7bit	all data must be "7 bit clean" (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
8bit	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see *fold_binary()* and *utf8* below for exceptions), but body parts may use the 8bit CTE.

A cte_type value of 8bit only works with BytesGenerator, not Generator, because strings cannot contain binary data. If a Generator is operating under a policy that specifies cte_type=8bit, it will act as if cte_type is 7bit.

raise_on_defect
If *True*, any defects encountered will be raised as errors. If *False* (the default), defects will be passed to the *register_defect()* method.

mangle_from_
If *True*, lines starting with "*From* " in the body are escaped by putting a > in front of them. This parameter is used when the message is being serialized by a generator. Default: *False*.

버전 3.5에 추가: The *mangle_from_* parameter.

message_factory
A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to None, in which case *Message* is used.

버전 3.6에 추가.

The following *Policy* method is intended to be called by code using the email library to create policy instances with custom settings:

clone(***kw*)
Return a new *Policy* instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining *Policy* methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(*obj*, *defect*)
Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of Defect.

The default implementation checks the *raise_on_defect* flag. If it is True, *defect* is raised as an exception. If it is False (the default), *obj* and *defect* are passed to *register_defect()*.

register_defect(*obj*, *defect*)
Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of Defect.

The default implementation calls the append method of the defects attribute of *obj*. When the email package calls *handle_defect*, *obj* will normally have a defects attribute that has an append method. Custom object types used with the email package (for example, custom Message objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(*name*)
Return the maximum allowed number of headers named *name*.

Called when a header is added to an *EmailMessage* or *Message* object. If the returned value is not 0 or None, and there are already a number of headers with the name *name* greater than or equal to the value returned, a *ValueError* is raised.

Because the default behavior of Message.__setitem__ is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in

the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

header_source_parse(*sourcelines*)
The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the ':' separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

header_store_parse(*name*, *value*)
The email package calls this method with the name and value provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

header_fetch_parse(*name*, *value*)
The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

fold(*name*, *value*)
The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header "folded" correctly (according to the policy settings) by composing the *name* with the *value* and inserting *linesep* characters at the appropriate places. See **RFC 5322** for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary(*name*, *value*)
The same as *fold()*, except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class email.policy.**EmailPolicy**(***kw*)
This concrete *Policy* provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) **RFC 5322**, **RFC 2047**, and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement **RFC 2047** and **RFC 5322**.

The default value for the *message_factory* attribute is *EmailMessage*.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

버전 3.6에 추가:[1]

utf8
If `False`, follow **RFC 5322**, supporting non-ASCII characters in headers by encoding them as "encoded words". If `True`, follow **RFC 6532** and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the `SMTPUTF8` extension (**RFC 6531**).

refold_source
If the value for a header in the `Message` object originated from a *parser* (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

`none`	all source values use original folding
`long`	source values that have any line that is longer than `max_line_length` will be refolded
`all`	all values are refolded.

The default is `long`.

header_factory
A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see *headerregistry*) is provided that supports custom parsing for the various address and date **RFC 5322** header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

content_manager
An object with at least two methods: get_content and set_content. When the *get_content()* or *set_content()* method of an *EmailMessage* object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to *raw_data_manager*.

버전 3.4에 추가.

The class provides the following concrete implementations of the abstract methods of *Policy*:

header_max_count(*name*)
Returns the value of the *max_count* attribute of the specialized class used to represent the header with the given name.

header_source_parse(*sourcelines*)
The name is parsed as everything up to the '`:`' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)
The name is returned unchanged. If the input value has a `name` attribute and it matches *name* ignoring case, the value is returned unchanged. Otherwise the *name* and *value* are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

[1] Originally added in 3.3 as a *provisional feature*.

header_fetch_parse(*name*, *value*)
If the value has a `name` attribute, it is returned to unmodified. Otherwise the *name*, and the *value* with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold(*name*, *value*)
Header folding is controlled by the *refold_source* policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the *name* and the *value* with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using *splitlines()*. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)
The same as *fold()* if *cte_type* is `7bit`, except that the returned value is bytes.

If *cte_type* is `8bit`, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of *EmailPolicy* provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the `HTTP` instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

email.policy.**default**
An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

email.policy.**SMTP**
Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

email.policy.**SMTPUTF8**
The same as `SMTP` except that *utf8* is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the *smtplib.SMTP.send_message()* method handles this automatically).

email.policy.**HTTP**
Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

email.policy.**strict**
Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing:

```
somepolicy + policy.strict
```

With all of these *EmailPolicies*, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a *Message* results in that header being parsed and a header object created.
- Fetching a header value from a *Message* results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the *EmailMessage* is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in *headerregistry*.

class email.policy.**Compat32**(**kw*)
This concrete *Policy* is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The *policy* module also defines an instance of this class, *compat32*, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the *Policy* default:

mangle_from_
The default is `True`.

The class provides the following concrete implementations of the abstract methods of *Policy*:

header_source_parse(*sourcelines*)
The name is parsed as everything up to the '`:`' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)
The name and value are returned unmodified.

header_fetch_parse(*name*, *value*)
If the value contains binary data, it is converted into a *Header* object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

fold(*name*, *value*)
Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)
Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

email.policy.**compat32**
An instance of *Compat32*, providing backward compatibility with the behavior of the email package in Python 3.2.

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**
add_payload()를 사용하여 페이 로드가 *Message* 객체에 추가되었지만, 페이 로드가 이미 스칼라(scalar)이고 메시지의 *Content-Type* 메인 유형이 *multipart*도 아니고 누락되지도 않았으면 발생합니다. *MultipartConversionError*는 *MessageError*와 내장 *TypeError*에서 다중 상속됩니다.

Message.add_payload()는 폐지되었으므로, 실제로 이 예외는 거의 발생하지 않습니다. 그러나 *MIMENonMultipart*에서 파생된 클래스(예를 들어 *MIMEImage*)의 인스턴스에서 *attach()* 메서드를 호출하면 예외가 발생할 수도 있습니다.

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.MessageDefect의 서브 클래스입니다.

- NoBoundaryInMultipartDefect – 메시지가 멀티 파트라고 주장했지만, *boundary* 파라미터가 없습니다.
- StartBoundaryNotFoundDefect – *Content-Type* 헤더에서 주장하는 시작 경계를 찾지 못했습니다.
- CloseBoundaryNotFoundDefect – 시작 경계가 발견되었지만, 해당하는 종료 경계가 발견되지 않았습니다.

  버전 3.3에 추가.
- FirstHeaderLineIsContinuationDefect – 메시지의 첫 번째 헤더 줄에 연장 줄(continuation line)이 있습니다.
- MisplacedEnvelopeHeaderDefect - 헤더 블록 중간에 “Unix From” 헤더가 있습니다.
- 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.

  버전 3.3에 추가.
- MalformedHeaderDefect – 콜론이 없거나 다른 식으로 잘못된 헤더가 발견되었습니다.

  버전 3.3부터 폐지: 이 결함은 여러 파이썬 버전에서 사용되지 않았습니다.
- MultipartInvariantViolationDefect – A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its *is_multipart()* method may return False even though its content type claims to be *multipart*.
- InvalidBase64PaddingDefect – base64로 인코딩된 바이트열 블록을 디코딩할 때, 패딩이 올바르지 않습니다. 디코딩을 수행하기 위해 충분한 패딩이 추가되지만, 바이트열을 디코딩한 결과는 유효하지 않을 수 있습니다.
- InvalidBase64CharactersDefect – base64로 인코딩된 바이트열 블록을 디코딩할 때, base64 알파벳 이외의 문자가 발견되었습니다. 문자는 무시되지만, 바이트열을 디코딩한 결과는 유효하지 않을 수 있습니다.

- `InvalidBase64LengthDefect` – base64로 인코딩된 바이트열 블록을 디코딩할 때, 비 패딩 base64 문자 수가 유효하지 않습니다 (4의 배수보다 1이 큽니다). 인코딩된 블록은 그대로 유지됩니다.

20.1.6 `email.headerregistry`: Custom Header Objects

Source code: Lib/email/headerregistry.py

버전 3.6에 추가:[1]

Headers are represented by customized subclasses of *str*. The particular class used to represent a given header is determined by the *header_factory* of the *policy* in effect when the headers are created. This section documents the particular `header_factory` implemented by the email package for handling **RFC 5322** compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from *EmailPolicy*, all headers are produced by *HeaderRegistry* and have *BaseHeader* as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class *UnstructuredHeader* as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the *HeaderRegistry*. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of *HeaderRegistry*, and finally the support classes used to represent the data parsed from structured headers.

class email.headerregistry.**BaseHeader**(*name*, *value*)

name and *value* are passed to `BaseHeader` from the *header_factory* call. The string value of any header object is the *value* fully decoded to unicode.

This base class defines the following read-only properties:

name

The name of the header (the portion of the field before the ':'). This is exactly the value passed in the *header_factory* call for *name*; that is, case is preserved.

defects

A tuple of `HeaderDefect` instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the *errors* module for a discussion of the types of defects that may be reported.

max_count

The maximum number of headers of this type that can have the same `name`. A value of `None` means unlimited. The `BaseHeader` value for this attribute is `None`; it is expected that specialized header classes will override this value as needed.

`BaseHeader` also provides the following method, which is called by the email library code and should not in general be called by application programs:

fold(*, *policy*)

Return a string containing *linesep* characters as required to correctly fold the header according to *policy*. A *cte_type* of `8bit` will be treated as if it were `7bit`, since headers may not contain arbitrary binary data. If *utf8* is `False`, non-ASCII data will be **RFC 2047** encoded.

[1] Originally added in 3.3 as a *provisional module*

BaseHeader by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, BaseHeader requires that the specialized class provide a *classmethod()* named parse. This method is called as follows:

```
parse(string, kwds)
```

kwds is a dictionary containing one pre-initialized key, defects. defects is an empty list. The parse method should append any detected defects to this list. On return, the kwds dictionary *must* contain values for at least the keys decoded and defects. decoded should be the string value for the header (that is, the header value fully decoded to unicode). The parse method should assume that *string* may contain content-transfer-encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

BaseHeader's __new__ then creates the header instance, and calls its init method. The specialized class only needs to provide an init method if it wishes to set additional attributes beyond those provided by BaseHeader itself. Such an init method should look like this:

```
def init(self, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

That is, anything extra that the specialized class puts in to the kwds dictionary should be removed and handled, and the remaining contents of kw (and args) passed to the BaseHeader init method.

class email.headerregistry.**UnstructuredHeader**

An "unstructured" header is the default type of header in **RFC 5322**. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the *Subject* header.

In **RFC 5322**, an unstructured header is a run of arbitrary text in the ASCII character set. **RFC 2047**, however, has an **RFC 5322** compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a *value* containing encoded words is passed to the constructor, the UnstructuredHeader parser converts such encoded words into unicode, following the **RFC 2047** rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

class email.headerregistry.**DateHeader**

RFC 5322 specifies a very specific format for dates within email headers. The DateHeader parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found "in the wild".

This header type provides the following additional attributes:

datetime

If the header value can be recognized as a valid date of one form or another, this attribute will contain a *datetime* instance representing that date. If the timezone of the input date is specified as -0000 (indicating it is in UTC but contains no information about the source timezone), then *datetime* will be a naive *datetime*. If a specific timezone offset is found (including *+0000*), then *datetime* will contain an aware datetime that uses *datetime.timezone* to record the timezone offset.

The decoded value of the header is determined by formatting the datetime according to the **RFC 5322** rules; that is, it is set to:

```
email.utils.format_datetime(self.datetime)
```

When creating a DateHeader, *value* may be *datetime* instance. This means, for example, that the following code is valid and does what one would expect:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of `-0000`. Much more useful is to use the *`localtime()`* function from the *`utils`* module:

```
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

class email.headerregistry.**AddressHeader**

Address headers are one of the most complex structured header types. The `AddressHeader` class provides a generic interface to any address header.

This header type provides the following additional attributes:

groups

A tuple of *`Group`* objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address `Groups` whose *`display_name`* is `None`.

addresses

A tuple of *`Address`* objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is "flattened" into a one dimensional list).

The `decoded` value of the header will have all encoded words decoded to unicode. *`idna`* encoded domain names are also decoded to unicode. The `decoded` value is set by *`join`*ing the *`str`* value of the elements of the `groups` attribute with `', '`.

A list of *`Address`* and *`Group`* objects in any combination may be used to set the value of an address header. `Group` objects whose `display_name` is `None` will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the `groups` attribute of the source header.

class email.headerregistry.**SingleAddressHeader**

A subclass of *`AddressHeader`* that adds one additional attribute:

address

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default *`policy`*), accessing this attribute will result in a *`ValueError`*.

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, *`max_count`* is set to 1.

class email.headerregistry.**MIMEVersionHeader**

There is really only one valid value for the *`MIME-Version`* header, and that is `1.0`. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per **RFC 2045**, then the header object will have non-`None` values for the following attributes:

version

The version number as a string, with any whitespace and/or comments removed.

major

The major version number as an integer

minor

The minor version number as an integer

class email.headerregistry.**ParameterizedMIMEHeader**

MIME headers all start with the prefix 'Content-'. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params

A dictionary mapping parameter names to parameter values.

class email.headerregistry.**ContentTypeHeader**
A *ParameterizedMIMEHeader* class that handles the *Content-Type* header.

content_type
The content type string, in the form `maintype/subtype`.

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**
A *ParameterizedMIMEHeader* class that handles the *Content-Disposition* header.

content-disposition
`inline` and `attachment` are the only valid values in common use.

class email.headerregistry.**ContentTransferEncoding**
Handles the *Content-Transfer-Encoding* header.

cte
Valid values are `7bit`, `8bit`, `base64`, and `quoted-printable`. See **RFC 2045** for more information.

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)
This is the factory used by *EmailPolicy* by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is `True` (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's `__bases__` list.

The default mappings are:

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

from UniqueAddressHeader

resent-from AddressHeader

reply-to UniqueAddressHeader

`HeaderRegistry` has the following methods:

map_to_type(*self*, *name*, *cls*)
name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__(*name*)
Construct and return a class to handle creating a *name* header.

__call__(*name*, *value*)
Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)
The class used to represent an email address. The general form of an address is:

```
[display_name] <username@domain>
```

or:

```
username@domain
```

where each part must conform to specific syntax rules spelled out in **RFC 5322**.

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not `Address` will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, unicode is *not* allowed in the username portion of the address.

display_name
The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username
The `username` portion of the address, with all quoting removed.

domain
The `domain` portion of the address.

addr_spec
The `username@domain` portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__()
The `str` value of the object is the address quoted according to **RFC 5322** rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (**RFC 5321**), `Address` handles one special case: if `username` and `domain` are both the empty string (or `None`), then the string value of the `Address` is `<>`.

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)
The class used to represent an address group. The general form of an address group is:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a `Group` may also be used to represent single addresses that are not part of a group by setting *display_name* to `None` and providing a list of the single address as *addresses*.

display_name
The `display_name` of the group. If it is `None` and there is exactly one `Address` in `addresses`, then the `Group` represents a single address that is not in a group.

addresses
A possibly empty tuple of *Address* objects representing the addresses in the group.

__str__()
The str value of a Group is formatted according to **RFC 5322**, but with no Content Transfer Encoding of any non-ASCII characters. If display_name is none and there is a single Address in the addresses list, the str value will be the same as the str of that single Address.

20.1.7 email.contentmanager: MIME 콘텐츠 관리

소스 코드: Lib/email/contentmanager.py

버전 3.6에 추가:[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_content는 응용 프로그램이 수동으로 디코딩할 필요 없이 유니코드 문자열을 반환하고, set_content는 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. multipart에서 호출되면 *KeyError*를 발생시킵니다. 파트가 text 파트이고 *errors*가 지정되면, 페이 로드를 유니코드로 디코딩할 때 에러 처리기로 사용합니다. 기본 에러 처리기는 replace입니다.

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.

- str의 경우, MIME maintype을 text로 설정하고, 서브 유형은 지정되었으면 *subtype*으로 설정하고, 지정되지 않았으면 plain으로 설정합니다.
- bytes의 경우, 지정된 *maintype*과 *subtype*을 사용하거나, 지정되지 않았으면 *TypeError*를 발생시킵니다.
- *EmailMessage* 객체의 경우, 메인 유형을 message로 설정하고, 서브 유형은 지정되었으면 *subtype*으로 설정하고, 지정되지 않았으면 rfc822로 설정합니다. *subtype*이 partial이면 에러를 발생시킵니다(bytes 객체를 사용하여 message/partial 파트를 구성해야 합니다).

*charset*이 제공되면(str에만 유효합니다), 지정된 문자 집합을 사용하여 문자열을 바이트열로 인코딩합니다. 기본값은 utf-8입니다. 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 binary입니다. 지정된 인코딩으로 입력을 인코딩할 수 없으면(예를 들어, 비 ASCII 값을 포함하는 입력에 대해 *cte*를 7bit로 지정합니다), *ValueError*를 발생시킵니다.

- `str` 객체의 경우, *cte*가 설정되지 않으면 휴리스틱을 사용하여 가장 간결한 인코딩을 결정합니다.
- *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`뿐입니다.

*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):

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
s.send_message(msg)
s.quit()
```

parser 모듈의 클래스를 사용하여 **RFC 822** 헤더를 쉽게 구문 분석할 수 있습니다:

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
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')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
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:

```
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# Open the files in binary mode.  Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                                    subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
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

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
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 = '.'
    # Create the message
    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'
```

(다음 페이지에 계속)

[1] 영감과 예를 주신 Matthew Dixon Cowles에게 감사드립니다.

(이전 페이지에서 계속)

```
    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            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)
    # Now send or store the message
    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

"""Unpack a MIME message into a directory of files."""

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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    with open(args.msgfile, 'rb') as fp:
        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/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                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

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
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!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cela ressemble à un excellent
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recipie
        </a> déjeuner.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
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

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
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()

# We can extract the richest alternative in order to display it:
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())
            # again strip the <> to go from email form of cid to html form.
            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:
    # The magic_html_parser has to rewrite the href="cid:...." attributes to
    # point to the filenames in partfiles.  It also has to do a safety-sanitize
    # of the html.  It could be written using html.parser.
    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)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

프롬프트까지, 위의 출력은 다음과 같습니다:

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!

Cela ressemble à un excellent recipie[1] déjeuner.
```

레거시 API:

20.1.9 email.message.Message: Representing an email message using the compat32 API

The *Message* class is very similar to the *EmailMessage* class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the *EmailMessage* class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for *Message*) policy *Compat32*. If you are going to use another policy, you should be using the *EmailMessage* class instead.

An email message consists of *headers* and a *payload*. Headers must be **RFC 5233** style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by a *Message* object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The *Message* pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of *Message* objects, for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Here are the methods of the *Message* class:

class email.message.**Message**(*policy=compat32*)

If *policy* is specified (it must be an instance of a *policy* class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the *policy* documentation.

버전 3.3에서 변경: The *policy* keyword argument was added.

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility reasons, *maxheaderlen* defaults to `0`, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a *Generator* instance and use its *flatten()* method directly. For example:

```
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()
```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode "unknown character" code points. (See also *as_bytes()* and *BytesGenerator*.)

버전 3.4에서 변경: the *policy* keyword argument was added.

__str__()
Equivalent to *as_string()*. Allows `str(msg)` to produce a string containing the formatted message.

as_bytes(*unixfrom=False*, *policy=None*)
Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a *BytesGenerator* instance and use its *flatten()* method directly. For example:

```
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()
```

버전 3.4에 추가.

__bytes__()
Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

버전 3.4에 추가.

is_multipart()
Return `True` if the message's payload is a list of sub-*Message* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that *is_multipart()* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *Message* is of type `message/rfc822`.)

set_unixfrom(*unixfrom*)
Set the message's envelope header to *unixfrom*, which should be a string.

get_unixfrom()
Return the message's envelope header. Defaults to `None` if the envelope header was never set.

attach(*payload*)
Add the given *payload* to the current payload, which must be `None` or a list of *Message* objects before the call. After the call, the payload will always be a list of *Message* objects. If you want to set the payload to a scalar object (e.g. a string), use *set_payload()* instead.

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*)
Return the current payload, which will be a list of *Message* objects when *is_multipart()* is `True`, or a string when *is_multipart()* is `False`. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, *get_payload()* will return the *i*-th element of the payload, counting from zero, if *is_multipart()* is `True`. An *IndexError* will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. *is_multipart()* is `False`) and *i* is given, a *TypeError* is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *Content-Transfer-Encoding* header. When `True` and the message is not a multipart, the payload will be decoded if this header's value is `quoted-printable` or `base64`. If some other encoding is used, or *Content-Transfer-Encoding* header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is `True`, then `None` is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (`InvalidBase64PaddingDefect` or `InvalidBase64CharactersDefect`, respectively).

When *decode* is `False` (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the `charset` specified by the *Content-Type* header, using the `replace` error handler. If no `charset` is specified, or if the `charset` given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *get_content()* and *iter_parts()*.

set_payload(*payload*, *charset=None*)
Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see *set_charset()* for details.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *set_content()*.

set_charset(*charset*)
Set the character set of the payload to *charset*, which can either be a *Charset* instance (see *email.charset*), a string naming a character set, or `None`. If it is a string, it will be converted to a *Charset* instance. If *charset* is `None`, the `charset` parameter will be removed from the *Content-Type* header (the message will not be otherwise modified). Anything else will generate a *TypeError*.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its `charset` parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified *Charset*, and a header with the appropriate value will be added. If a

Content-Transfer-Encoding header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *charset* parameter of the `email.emailmessage.EmailMessage.set_content()` method.

get_charset()
Return the *Charset* instance associated with the message's payload.

This is a legacy method. On the `EmailMessage` class it always returns `None`.

The following methods implement a mapping-like interface for accessing the message's **RFC 2822** headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in a *Message* object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as *Header* objects with a charset of *unknown-8bit*.

__len__()
Return the total number of headers, including duplicates.

__contains__(*name*)
Return `True` if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the `in` operator, e.g.:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant named headers.

__setitem__(*name*, *val*)
Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)
Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)
Return the value of the named header field. This is identical to *__getitem__()* except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods:

get_all(*name*, *failobj=None*)
Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)
Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here's an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)
Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a *KeyError* is raised.

get_content_type()
Return the message's content type. The returned string is coerced to lower case of the form *maintype/subtype*. If there was no *Content-Type* header in the message the default type as given by *get_default_type()* will be returned. Since according to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value.

RFC 2045 defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.

get_content_maintype()
Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()
Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()
Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)
Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header.

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)
Return the message's *Content-Type* parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the `'='` sign. The left hand side of the `'='` is the key, while the right hand side is the value. If there is no `'='` sign in the parameter the value is the empty string, otherwise the value is as described in *get_param()* and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no *Content-Type* header. Optional *header* is the header to search instead of *Content-Type*.

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*)
Return the value of the *Content-Type* header's parameter *param* as a string. If the message has no *Content-Type* header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of *Content-Type*.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

If your application doesn't care whether the parameter was encoded as in **RFC 2231**, you can collapse the parameter value by calling *email.utils.collapse_rfc2231_value()*, passing in the return value from *get_param()*. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the `VALUE` item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

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*)
Set a parameter in the *Content-Type* header. If the parameter already exists in the header, its value will be replaced with *value*. If the *Content-Type* header as not yet been defined for this message, it will be set to *text/plain* and the new parameter value will be appended as per **RFC 2045**.

Optional *header* specifies an alternative header to `Content-Type`, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to **RFC 2231**. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

버전 3.4에서 변경: `replace` keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)
Remove the given parameter completely from the `Content-Type` header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to `Content-Type`.

set_type(*type*, *header='Content-Type'*, *requote=True*)
Set the main type and subtype for the `Content-Type` header. *type* must be a string in the form `maintype/subtype`, otherwise a `ValueError` is raised.

This method replaces the `Content-Type` header, keeping all the parameters in place. If *requote* is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the `Content-Type` header is set a `MIME-Version` header is also added.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj=None*)
Return the value of the `filename` parameter of the `Content-Disposition` header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the `Content-Type` header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per `email.utils.unquote()`.

get_boundary(*failobj=None*)
Return the value of the `boundary` parameter of the `Content-Type` header of the message, or *failobj* if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per `email.utils.unquote()`.

set_boundary(*boundary*)
Set the `boundary` parameter of the `Content-Type` header to *boundary*. `set_boundary()` will always quote *boundary* if necessary. A `HeaderParseError` is raised if the message object has no `Content-Type` header.

Note that using this method is subtly different than deleting the old `Content-Type` header and adding a new one with the new boundary via `add_header()`, because `set_boundary()` preserves the order of the `Content-Type` header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original `Content-Type` header.

get_content_charset(*failobj=None*)
Return the `charset` parameter of the `Content-Type` header, coerced to lower case. If there is no `Content-Type` header, or if that header has no `charset` parameter, *failobj* is returned.

Note that this method differs from `get_charset()` which returns the `Charset` instance for the default encoding of the message body.

get_charsets(*failobj=None*)
Return a list containing the character set names in the message. If the message is a `multipart`, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *`Content-Type`* header for the represented subpart. However, if the subpart has no *`Content-Type`* header, no `charset` parameter, or is not of the *`text`* main MIME type, then that item in the returned list will be *failobj*.

`get_content_disposition``()`
Return the lowercased value (without parameters) of the message's *`Content-Disposition`* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183**.

버전 3.5에 추가.

`walk``()`
The *`walk()`* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *`walk()`* as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> 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
```

`walk` iterates over the subparts of any part where *`is_multipart()`* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> 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
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

`Message` objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

`preamble`
The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it

falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the `Parser` discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the `Generator` is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue
The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the `Generator` to print a newline at the end of the file.

defects
The *defects* attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

20.1.10 `email.mime`: Creating email and MIME objects from scratch

Source code: Lib/email/mime/

This module is part of the legacy (`Compat32`) email API. Its functionality is partially replaced by the `contentmanager` in the new API, but in certain applications these classes may still be useful, even in non-legacy code.

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual `Message` objects by hand. In fact, you can also take an existing structure and add new `Message` objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating `Message` instances, adding attachments and all the appropriate headers manually. For MIME messages though, the `email` package provides some convenient subclasses to make things easier.

Here are the classes:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)
Module: `email.mime.base`

This is the base class for all the MIME-specific subclasses of `Message`. Ordinarily you won't create instances specifically of `MIMEBase`, although you could. `MIMEBase` is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the `Content-Type` major type (e.g. `text` or `image`), and *_subtype* is the `Content-Type` minor type (e.g. `plain` or `gif`). *_params* is a parameter key/value dictionary and is passed directly to `Message.add_header`.

If *policy* is specified, (defaults to the `compat32` policy) it will be passed to `Message`.

The `MIMEBase` class always adds a `Content-Type` header (based on *_maintype*, *_subtype*, and *_params*), and a `MIME-Version` header (always set to `1.0`).

버전 3.6에서 변경: Added *policy* keyword-only parameter.

class email.mime.nonmultipart.**MIMENonMultipart**
Module: email.mime.nonmultipart

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are not *multipart*. The primary purpose of this class is to prevent the use of the *attach()* method, which only makes sense for *multipart* messages. If *attach()* is called, a *MultipartConversionError* exception is raised.

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, ***, *policy=compat32*, ***_params*)
Module: email.mime.multipart

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are *multipart*. Optional *_subtype* defaults to *mixed*, but can be used to specify the subtype of the message. A *Content-Type* header of *multipart/_subtype* will be added to the message object. A *MIME-Version* header will also be added.

Optional *boundary* is the multipart boundary string. When None (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the *Message.attach* method.

Optional *policy* argument defaults to *compat32*.

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

버전 3.6에서 변경: Added *policy* keyword-only parameter.

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)
Module: email.mime.application

A subclass of *MIMENonMultipart*, the *MIMEApplication* class is used to represent MIME message objects of major type *application*. *_data* is a string containing the raw byte data. Optional *_subtype* specifies the MIME subtype and defaults to *octet-stream*.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the *MIMEApplication* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the base class constructor.

버전 3.6에서 변경: Added *policy* keyword-only parameter.

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)
Module: email.mime.audio

A subclass of *MIMENonMultipart*, the *MIMEAudio* class is used to create MIME message objects of major type *audio*. *_audiodata* is a string containing the raw audio data. If this data can be decoded by the standard Python module *sndhdr*, then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the audio subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the *MIMEAudio* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any

Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the base class constructor.

버전 3.6에서 변경: Added *policy* keyword-only parameter.

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

Module: email.mime.image

A subclass of *MIMENonMultipart*, the *MIMEImage* class is used to create MIME message objects of major type *image*. *_imagedata* is a string containing the raw image data. If this data can be decoded by the standard Python module *imghdr*, then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the image subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the *MIMEImage* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the *MIMEBase* constructor.

버전 3.6에서 변경: Added *policy* keyword-only parameter.

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

Module: email.mime.message

A subclass of *MIMENonMultipart*, the *MIMEMessage* class is used to create MIME objects of main type *message*. *_msg* is used as the payload, and must be an instance of class *Message* (or a subclass thereof), otherwise a *TypeError* is raised.

Optional *_subtype* sets the subtype of the message; it defaults to *rfc822*.

Optional *policy* argument defaults to *compat32*.

버전 3.6에서 변경: Added *policy* keyword-only parameter.

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)

Module: email.mime.text

A subclass of *MIMENonMultipart*, the *MIMEText* class is used to create MIME objects of major type *text*. *_text* is the string for the payload. *_subtype* is the minor type and defaults to *plain*. *_charset* is the character set of the text and is passed as an argument to the *MIMENonMultipart* constructor; it defaults to us-ascii if the string contains only ascii code points, and utf-8 otherwise. The *_charset* parameter accepts either a string or a *Charset* instance.

Unless the *_charset* argument is explicitly set to None, the MIMEText object created will have both a *Content-Type* header with a charset parameter, and a *Content-Transfer-Encoding* header. This means that a subsequent set_payload call will not result in an encoded payload, even if a charset is passed in the set_payload command. You can "reset" this behavior by deleting the Content-Transfer-Encoding header, after which a set_payload call will automatically encode the new payload (and add a new *Content-Transfer-Encoding* header).

Optional *policy* argument defaults to *compat32*.

버전 3.5에서 변경: *_charset* also accepts *Charset* instances.

버전 3.6에서 변경: Added *policy* keyword-only parameter.

20.1.11 email.header: Internationalized headers

Source code: Lib/email/header.py

This module is part of the legacy (Compat32) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the *EmailMessage* class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

The remaining text in this section is the original documentation of the module.

RFC 2822 is the base standard that describes the format of email messages. It derives from the older **RFC 822** standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into **RFC 2822**-compliant format. These RFCs include **RFC 2045**, **RFC 2046**, **RFC 2047**, and **RFC 2231**. The *email* package supports these standards in its *email.header* and *email.charset* modules.

If you want to include non-ASCII characters in your email headers, say in the *Subject* or *To* fields, you should use the *Header* class and assign the field in the *Message* object to an instance of *Header* instead of using a string for the header value. Import the *Header* class from the *email.header* module. For example:

```
>>> 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'
```

Notice here how we wanted the *Subject* field to contain a non-ASCII character? We did this by creating a *Header* instance and passing in the character set that the byte string was encoded in. When the subsequent *Message* instance was flattened, the *Subject* field was properly **RFC 2047** encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

Here is the *Header* class description:

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

Create a MIME-compliant header that can contain strings in different character sets.

Optional *s* is the initial header value. If None (the default), the initial header value is not set. You can later append to the header with *append()* method calls. *s* may be an instance of *bytes* or *str*, but see the *append()* documentation for semantics.

Optional *charset* serves two purposes: it has the same meaning as the *charset* argument to the *append()* method. It also sets the default character set for all subsequent *append()* calls that omit the *charset* argument. If *charset* is not provided in the constructor (the default), the us-ascii character set is used both as *s*'s initial charset and as the default for subsequent *append()* calls.

The maximum line length can be specified explicitly via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn't included in *s*, e.g. *Subject*) pass in the name of the field in *header_name*. The default *maxlinelen* is 76, and the default value for *header_name* is None, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be **RFC 2822**-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. *continuation_ws* defaults to a single space character.

Optional *errors* is passed straight through to the `append()` method.

append(*s*, *charset=None*, *errors='strict'*)
Append the string *s* to the MIME header.

Optional *charset*, if given, should be a `Charset` instance (see `email.charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the *charset* given in the constructor is used.

s may be an instance of `bytes` or `str`. If it is an instance of `bytes`, then *charset* is the encoding of that byte string, and a `UnicodeError` will be raised if the string cannot be decoded with that character set.

If *s* is an instance of `str`, then *charset* is a hint specifying the character set of the characters in the string.

In either case, when producing an **RFC 2822**-compliant header using **RFC 2047** rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional *errors* is passed as the errors argument to the decode call if *s* is a byte string.

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)
Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional *splitchars* is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of **RFC 2822**'s 'higher level syntactic breaks': split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect **RFC 2047** encoded lines.

maxlinelen, if given, overrides the instance's value for the maximum line length.

linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (`\n`), but `\r\n` can be specified in order to produce headers with RFC-compliant line separators.

버전 3.2에서 변경: Added the *linesep* argument.

The `Header` class also provides a number of methods to support standard operators and built-in functions.

__str__()
Returns an approximation of the `Header` as a string, using an unlimited line length. All pieces are converted to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of `'unknown-8bit'` are decoded as ASCII using the `'replace'` error handler.

버전 3.2에서 변경: Added handling for the `'unknown-8bit'` charset.

__eq__(*other*)
This method allows you to compare two `Header` instances for equality.

__ne__(*other*)
This method allows you to compare two `Header` instances for inequality.

The `email.header` module also provides the following convenient functions.

email.header.**decode_header**(*header*)
Decode a message header value without converting the character set. The header value is in *header*.

This function returns a list of `(decoded_string, charset)` pairs containing each of the decoded parts of the header. *charset* is `None` for non-encoded parts of the header, otherwise a lower case string containing the name of the character set specified in the encoded string.

Here's an example:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

`email.header.`**`make_header`**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)

Create a *Header* instance from a sequence of pairs as returned by *decode_header()*.

decode_header() takes a header value string and returns a sequence of pairs of the format `(decoded_string, charset)` where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a *Header* instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the *Header* constructor.

20.1.12 `email.charset`: Representing character sets

Source code: Lib/email/charset.py

This module is part of the legacy (`Compat32`) email API. In the new API only the aliases table is used.

The remaining text in this section is the original documentation of the module.

This module provides a class *Charset* for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of *Charset* are used in several other modules within the *email* package.

Import this class from the *email.charset* module.

class `email.charset.`**`Charset`**(*input_charset=DEFAULT_CHARSET*)

Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional *input_charset* is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if *input_charset* is `iso-8859-1`, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If *input_charset* is `euc-jp`, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the `euc-jp` character set to the `iso-2022-jp` character set.

Charset instances have the following data attributes:

`input_charset`

The initial character set specified. Common aliases are converted to their *official* email names (e.g. `latin_1` is converted to `iso-8859-1`). Defaults to 7-bit `us-ascii`.

header_encoding
If the character set must be encoded before it can be used in an email header, this attribute will be set to `Charset.QP` (for quoted-printable), `Charset.BASE64` (for base64 encoding), or `Charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding
Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. `Charset.SHORTEST` is not allowed for *body_encoding*.

output_charset
Some character sets must be converted before they can be used in email headers or bodies. If the *input_charset* is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be `None`.

input_codec
The name of the Python codec used to convert the *input_charset* to Unicode. If no conversion codec is necessary, this attribute will be `None`.

output_codec
The name of the Python codec used to convert Unicode to the *output_charset*. If no conversion codec is necessary, this attribute will have the same value as the *input_codec*.

Charset instances also have the following methods:

get_body_encoding()
Return the content transfer encoding used for body encoding.

This is either the string `quoted-printable` or `base64` depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the *Content-Transfer-Encoding* header itself to whatever is appropriate.

Returns the string `quoted-printable` if *body_encoding* is `QP`, returns the string `base64` if *body_encoding* is `BASE64`, and returns the string `7bit` otherwise.

get_output_charset()
Return the output character set.

This is the *output_charset* attribute if that is not `None`, otherwise it is *input_charset*.

header_encode(*string*)
Header-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *header_encoding* attribute.

header_encode_lines(*string*, *maxlengths*)
Header-encode a *string* by converting it first to bytes.

This is similar to *header_encode()* except that the string is fit into maximum line lengths as given by the argument *maxlengths*, which must be an iterator: each element returned from this iterator will provide the next maximum line length.

body_encode(*string*)
Body-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *body_encoding* attribute.

The *Charset* class also provides a number of methods to support standard operations and built-in functions.

__str__()
Returns *input_charset* as a string coerced to lower case. `__repr__()` is an alias for *__str__()*.

__eq__(*other*)
This method allows you to compare two *Charset* instances for equality.

__ne__(*other*)
This method allows you to compare two *Charset* instances for inequality.

The *email.charset* module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)
Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional *header_enc* and *body_enc* is either `Charset.QP` for quoted-printable, `Charset.BASE64` for base64 encoding, `Charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for *header_enc*. The default is `None` for no encoding.

Optional *output_charset* is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method `Charset.convert()` is called. The default is to output in the same character set as the input.

Both *input_charset* and *output_charset* must have Unicode codec entries in the module's character set-to-codec mapping; use *add_codec()* to add codecs the module does not know about. See the *codecs* module's documentation for more information.

The global character set registry is kept in the module global dictionary `CHARSETS`.

email.charset.**add_alias**(*alias*, *canonical*)
Add a character set alias. *alias* is the alias name, e.g. `latin-1`. *canonical* is the character set's canonical name, e.g. `iso-8859-1`.

The global charset alias registry is kept in the module global dictionary `ALIASES`.

email.charset.**add_codec**(*charset*, *codecname*)
Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. *codecname* is the name of a Python codec, as appropriate for the second argument to the *str*'s *encode()* method.

20.1.13 email.encoders: Encoders

Source code: Lib/email/encoders.py

This module is part of the legacy (`Compat32`) email API. In the new API the functionality is provided by the *cte* parameter of the *set_content()* method.

This module is deprecated in Python 3. The functions provided here should not be called explicitly since the *MIMEText* class sets the content type and CTE header using the *_subtype* and *_charset* values passed during the instantiation of that class.

The remaining text in this section is the original documentation of the module.

When creating *Message* objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for *image/** and *text/** type messages containing binary data.

The *email* package provides some convenient encodings in its `encoders` module. These encoders are actually used by the *MIMEAudio* and *MIMEImage* class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the *Content-Transfer-Encoding* header as appropriate.

Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a *TypeError* if passed a message whose type is multipart.

Here are the encoding functions provided:

email.encoders.**encode_quopri**(*msg*)
Encodes the payload into quoted-printable form and sets the *Content-Transfer-Encoding* header to quoted-printable[1]. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

email.encoders.**encode_base64**(*msg*)
Encodes the payload into base64 form and sets the *Content-Transfer-Encoding* header to base64. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

email.encoders.**encode_7or8bit**(*msg*)
This doesn't actually modify the message's payload, but it does set the *Content-Transfer-Encoding* header to either 7bit or 8bit as appropriate, based on the payload data.

email.encoders.**encode_noop**(*msg*)
This does nothing; it doesn't even set the *Content-Transfer-Encoding* header.

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*)
지역 시간을 어웨어 datetime 객체로 반환합니다. 인자 없이 호출되면, 현재 시각을 반환합니다. 그렇지 않으면 *dt* 인자가 *datetime* 인스턴스여야 하며, 시스템 시간대 데이터베이스에 따라 지역 시간대로 변환됩니다. *dt*가 나이브하면 (즉, dt.tzinfo가 None이면), 지역 시간으로 간주합니다. 이 경우, *isdst* 에 대한 양수나 0 값은 localtime 이 지정된 시간에 서머 타임(예를 들어, 일광 절약 시간)이 유효한지 그렇지 않은지를 처음에 가정하게 합니다. *isdst*에 대한 음수 값은 localtime 이 서머 타임이 지정된 시간에 유효한지를 결정하게 합니다.

버전 3.3에 추가.

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.

[1] Note that encoding with *encode_quopri()* also encodes all tabs and space characters in the data.

email.utils.**parseaddr**(*address*)
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.

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*)
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. 다음은 메시지의 모든 수신자를 얻는 간단한 예입니다:

```
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)
```

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*)
Performs the same function as *parsedate()*, but returns either None or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[1]. If the input string has no timezone, the last element of the tuple returned is 0, which represents UTC. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_to_datetime**(*date*)
*format_datetime()*의 역. *parsedate()*와 같은 기능을 수행하지만, 성공 시에 *datetime*을 반환합니다. 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.

버전 3.3에 추가.

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.

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)
RFC 2822에 따르는 날짜 문자열을 반환합니다, 예를 들어:

[1] 시간대 오프셋의 부호는 같은 시간대에 대한 time.timezone 변수의 부호와 반대임에 유의하십시오; 이 모듈이 **RFC 2822**를 따르지만, 후자의 변수는 POSIX 표준을 따릅니다.

```
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.

버전 3.3에 추가.

`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 *nntplib* NNTP (Net News Transport Protocol) 클라이언트

모듈 *mailbox* 다양한 표준 형식을 사용하여 디스크에 메시지 모음을 만들고, 읽고, 관리하는 도구.

모듈 *smtpd* SMTP 서버 프레임워크 (주로 테스트에 유용합니다)

20.2 json — JSON 인코더와 디코더

소스 코드: Lib/json/__init__.py

RFC 7159(**RFC 4627**을 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 아닙니다).

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.

*json*은 표준 라이브러리 *marshal*과 *pickle* 모듈 사용자에게 익숙한 API를 제공합니다.

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({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

JSON 디코딩:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

(다음 페이지에 계속)

[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.

(이전 페이지에서 계속)

```
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

JSON 객체 디코딩 특수화:

```
>>> 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]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, 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', ']']
```

셸에서 *json.tool*을 사용하여 유효성을 검사하고 예쁘게 인쇄합니다:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

자세한 설명은 명령 줄 인터페이스를 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.

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 변환표를 사용하여 *obj*를 JSON 형식 스트림으로 *fp*(`.write()`를 지원하는 파일류 객체)로 직렬화합니다.

*skipkeys*가 참이면 (기본값: `False`), 기본형(*str*, *int*, *float*, *bool*, `None`)이 아닌 딕셔너리 키는 *TypeError*를 발생시키는 대신 건너뜁니다.

json 모듈은 항상 *bytes* 객체가 아니라 *str* 객체를 생성합니다. 따라서, `fp.write()`는 *str* 입력을 지원해야 합니다.

*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 거짓이면 (기본값: `True`), 컨테이너형에 대한 순환 참조 검사를 건너뛰고 순환 참조는 *OverflowError*를 일으킵니다 (또는 더 나빠질 수 있습니다).

*allow_nan*이 거짓이면 (기본값: `True`), JSON 사양을 엄격히 준수하여 범위를 벗어난 *float* 값(`nan`, `inf`, `-inf`)을 직렬화하면 *ValueError*를 일으킵니다. *allow_nan*이 참이면, JavaScript의 대응 물(`NaN`, `Infinity`, `-Infinity`)이 사용됩니다.

*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.

*sort_keys*가 참이면 (기본값: `False`), 딕셔너리의 출력이 키로 정렬됩니다.

사용자 정의 *JSONEncoder* 서브 클래스(예를 들어, `default()` 메서드를 재정의하여 추가 형을 직렬화하는 것)를 사용하려면, *cls* 키워드 인자로 지정하십시오; 그렇지 않으면 *JSONEncoder*가 사용됩니다.

버전 3.6에서 변경: 모든 선택적 매개 변수는 이제 키워드-전용입니다.

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.

---

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*)

이 변환표를 사용하여 *fp*(JSON 문서를 포함하는 `.read()`를 지원하는 텍스트 파일이나 바이너리 파일)를 파이썬 객체로 역 직렬화합니다.

*object_hook*은 모든 오브젝트 리터럴의 디코딩된 결과(*dict*)로 호출되는 선택적 함수입니다. *object_hook*의 반환 값이 *dict* 대신에 사용됩니다. 이 기능은 사용자 정의 디코더를 구현하는 데 사용할 수 있습니다 (예를 들어, JSON-RPC 클래스 힌팅(class hinting)).

*object_pairs_hook*은 모든 오브젝트 리터럴의 쌍의 순서 있는 목록으로 디코딩된 결과로 호출되는 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*가 지정되면, 디코딩될 모든 JSON float의 문자열로 호출됩니다. 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*가 지정되면, 디코딩될 모든 JSON int의 문자열로 호출됩니다. 기본적으로 이것은 `int(num_str)`와 동등합니다. JSON 정수에 대해 다른 데이터형이나 구문 분석기를 사용하고자 할 때 사용될 수 있습니다 (예를 들어 *float*).

버전 3.7.14에서 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.

*parse_constant*가 지정되면, 다음과 같은 문자열 중 하나로 호출됩니다: `'-Infinity'`, `'Infinity'`, `'NaN'`. 잘못된 JSON 숫자를 만날 때 예외를 발생시키는 데 사용할 수 있습니다.

버전 3.1에서 변경: *parse_constant*는 더는 'null', 'true', 'false'에 대해 호출되지 않습니다.

사용자 정의 *JSONDecoder* 서브 클래스를 사용하려면, `cls` 키워드 인자로 지정하십시오; 그렇지 않으면 *JSONDecoder*가 사용됩니다. 추가 키워드 인자는 클래스 생성자에 전달됩니다.

역 직렬화되는 데이터가 유효한 JSON 문서가 아니면, *JSONDecodeError*가 발생합니다.

버전 3.6에서 변경: 모든 선택적 매개 변수는 이제 키워드-전용입니다.

버전 3.6에서 변경: *fp*는 이제 바이너리 파일이 될 수 있습니다. 입력 인코딩은 UTF-8, UTF-16 또는 UTF-32 여야 합니다.

json.**loads**(*s*, *, *encoding=None*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

이 변환표를 사용하여 *s*(JSON 문서를 포함하는 *str*, *bytes* 또는 *bytearray* 인스턴스)를 파이썬 객체로 역 직렬화합니다.

무시되고 폐지된 *encoding*을 제외하고는, 다른 인자는 *load()*와 같은 의미를 가집니다.

역 직렬화되는 데이터가 유효한 JSON 문서가 아니면, *JSONDecodeError*가 발생합니다.

버전 3.6에서 변경: *s*는 이제 *bytes*나 *bytearray* 형일 수 있습니다. 입력 인코딩은 UTF-8, UTF-16 또는 UTF-32 여야 합니다.

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*이 지정되면, 모든 JSON 오브젝트의 디코딩된 결과로 호출되며, 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*이 지정되면 모든 오브젝트 리터럴의 쌍의 순서 있는 목록으로 디코딩된 결과로 호출됩니다. *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*가 지정되면, 디코딩될 모든 JSON float의 문자열로 호출됩니다. 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*가 지정되면, 디코딩될 모든 JSON int의 문자열로 호출됩니다. 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*가 지정되면, 다음과 같은 문자열 중 하나로 호출됩니다: `'-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 o에 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* 또는 None이 아닌 키의 인코딩을 시도하는 것은 *TypeError*입니다. *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(*OverflowError*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 -Infinity는 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 None이면 (', ', ': ')이고, 그렇지 않으면 (',', ': ')입니다. 가장 간결한 JSON 표현을 얻으려면, (',', ':')를 지정하여 공백을 제거해야 합니다.

버전 3.4에서 변경: *indent*가 None이 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 세비 클래스에 이 메서드를 구현하십시오.

예를 들어, 임의의 이터레이터를 지원하려면, 다음과 같이 default를 구현할 수 있습니다:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return json.JSONEncoder.default(self, 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.

버전 3.5에 추가.

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 매개변수는 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:

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-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 명령 줄 인터페이스

소스 코드: Lib/json/tool.py

---

json.tool 모듈은 JSON 객체의 유효성을 검사하고 예쁘게 인쇄하는 간단한 명령 줄 인터페이스를 제공합니다.

선택적 `infile`과 `outfile` 인자가 지정되지 않으면, 각각 *sys.stdin*과 *sys.stdout*이 사용됩니다:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
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.

명령 줄 옵션

infile
유효성을 검사하거나 예쁘게 인쇄할 JSON 파일:

```
$ python -m json.tool 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.

버전 3.5에 추가.

-h, --help
도움말 메시지를 표시합니다.

20.3 mailcap — Mailcap 파일 처리

소스 코드: Lib/mailcap.py

Mailcap 파일은 메일 리더와 웹 브라우저와 같은 MIME 인식 응용 프로그램이 MIME 형식의 파일에 따라 반응하는 방식을 구성하는 데 사용됩니다. ("mailcap" 이라는 이름은 "mail capability" 라는 구문에서 왔습니다.) 예를 들어, mailcap 파일에는 `video/mpeg; xmpeg %s`와 같은 줄이 있을 수 있습니다. 그러면, 사용자가 MIME 유형이 *video/mpeg* 인 전자 메일 메시지 또는 웹 문서를 만나면, `%s`가 파일명(대개 임시 파일에 속한 파일)으로 치환되고 **xmpeg** 프로그램을 자동으로 시작하여 파일을 볼 수 있습니다.

The mailcap format is documented in **RFC 1524**, "A User Agent Configuration Mechanism For Multimedia Mail Format Information", but is not an Internet standard. However, mailcap files are supported on most Unix systems.

mailcap.**findmatch**(*caps*, *MIMEtype*, *key='view'*, *filename='/dev/null'*, *plist=[]*)
2-튜플을 반환합니다; 첫 번째 요소는 실행될 명령 줄(*os.system()*에 전달될 수 있음)을 포함하는 문자열이고, 두 번째 요소는 지정된 MIME 유형에 대한 mailcap 항목입니다. 일치하는 MIME 유형을 찾을 수 없으면 `(None, None)`이 반환됩니다.

*key*는 원하는 이름인데, 수행할 활동의 유형을 나타냅니다; 가장 흔히 MIME 형식의 데이터 본문을 보고만 싶으므로 기본값은 'view' 입니다. 주어진 MIME 유형의 새 본문을 만들거나 기존 본문 데이터를 변경하려고 할 때, 다른 가능한 값으로 'compose' 이나 'edit' 가 있습니다. 이 필드의 전체 목록은 **RFC 1524**를 참조하십시오.

*filename*은 명령 줄에서 `%s`에 치환될 파일명입니다. 기본값은 `'/dev/null'` 이지만, 거의 확실하게 원하는 것이 아닐 것이기 때문에, 보통 파일명을 지정하여 이를 대체합니다.

*plist*는 이름있는 매개 변수를 포함하는 리스트일 수 있습니다; 기본값은 단순히 빈 리스트입니다. 리스트의 각 항목은 매개 변수 이름, 등호(`'='`) 및 매개 변수의 값이 포함된 문자열이어야 합니다. Mailcap 항목은 `%{foo}`와 같은 이름있는 매개 변수를 포함할 수 있는데, 'foo' 라는 이름의 매개 변수의 값으로 치환됩니다. 예를 들어, 명령 줄 `showpartial %{id} %{number} %{total}`이 mailcap 파일에 있고, *plist*가 `['id=1', 'number=2', 'total=3']`로 설정되었으면, 결과 명령 줄은 `'showpartial 1 2 3'`가 됩니다.

mailcap 파일에서, "test" 필드를 선택적으로 지정하여 mailcap 줄이 적용되는지를 판단하기 위해 일부 외부 조건(가령 기계 아키텍처나 사용 중인 윈도우 시스템)을 검사할 수 있습니다. *findmatch()*는 자동으로 그러한 조건을 검사하고 검사가 실패하면 항목을 건너뜁니다.

버전 3.7.16에서 변경: To prevent security issues with shell metacharacters (symbols that have special effects in a shell command line), `findmatch` will refuse to inject ASCII characters other than alphanumerics and `@+=:,./-_` into the returned command line.

If a disallowed character appears in *filename*, `findmatch` will always return `(None, None)` as if no entry was found. If such a character appears elsewhere (a value in *plist* or in *MIMEtype*), `findmatch` will ignore all mailcap entries which use that value. A *warning* will be raised in either case.

`mailcap.`**`getcaps`**`()`

MIME 유형을 mailcap 파일 항목 리스트에 매핑하는 딕셔너리를 반환합니다. 이 딕셔너리는 *findmatch()* 함수에 전달되어야 합니다. 항목은 딕셔너리의 리스트로 저장되지만, 이 표현의 세부 사항을 알 필요는 없습니다.

이 정보는 시스템에서 발견된 모든 mailcap 파일에서 파생됩니다. 사용자의 mailcap 파일 `$HOME/.mailcap`의 설정은 시스템 mailcap 파일 `/etc/mailcap`, `/usr/etc/mailcap` 및 `/usr/local/etc/mailcap`의 설정보다 우선합니다.

사용 예:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

20.4 `mailbox` — Manipulate mailboxes in various formats

Source code: Lib/mailbox.py

This module defines two classes, *Mailbox* and *Message*, for accessing and manipulating on-disk mailboxes and the messages they contain. *Mailbox* offers a dictionary-like mapping from keys to messages. *Message* extends the *email.message* module's *Message* class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

더 보기:

Module *email* Represent and manipulate messages.

20.4.1 Mailbox objects

class mailbox.**Mailbox**
A mailbox, which may be inspected and modified.

The *Mailbox* class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from *Mailbox* and your code should instantiate a particular subclass.

The *Mailbox* interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the *Mailbox* instance with which they will be used and are only meaningful to that *Mailbox* instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a *Mailbox* instance using the set-like method *add()* and removed using a `del` statement or the set-like methods *remove()* and *discard()*.

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a *Message* instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a *Mailbox* instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the *Mailbox* instance.

The default *Mailbox* iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a *KeyError* exception if the corresponding message is subsequently removed.

> **경고:** Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the *lock()* and *unlock()* methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

Mailbox instances have the following methods:

add(*message*)
Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a *Message* instance, an *email.message.Message* instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific *Message* subclass (e.g., if it's an *mboxMessage* instance and this is an *mbox* instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

버전 3.2에서 변경: Support for binary input was added.

remove(*key*)
__delitem__(*key*)
discard(*key*)
Delete the message corresponding to *key* from the mailbox.

If no such message exists, a *KeyError* exception is raised if the method was called as *remove()* or *__delitem__()* but no exception is raised if the method was called as *discard()*. The behavior of *discard()* may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(*key*, *message*)
Replace the message corresponding to *key* with *message*. Raise a `KeyError` exception if no message already corresponds to *key*.

As with `add()`, parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

iterkeys()
keys()
Return an iterator over all keys if called as `iterkeys()` or return a list of keys if called as `keys()`.

itervalues()
__iter__()
values()
Return an iterator over representations of all messages if called as `itervalues()` or `__iter__()` or return a list of such representations if called as `values()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

참고: The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

iteritems()
items()
Return an iterator over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation, if called as `iteritems()` or return a list of such pairs if called as `items()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get(*key*, *default=None*)
__getitem__(*key*)
Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(*key*)
Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_bytes(*key*)
Return a byte representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists.

버전 3.2에 추가.

get_string(*key*)
Return a string representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The message is processed through `email.message.Message` to convert it to a 7bit clean representation.

get_file(*key*)
Return a file-like representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

버전 3.2에서 변경: The file object really is a binary file; previously it was incorrectly returned in text mode. Also, the file-like object now supports the context management protocol: you can use a `with` statement to automatically close it.

참고: Unlike other representations of messages, file-like representations are not necessarily independent of the *Mailbox* instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

__contains__(*key*)
Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()
Return a count of messages in the mailbox.

clear()
Delete all messages from the mailbox.

pop(*key*, *default=None*)
Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default*. The message is represented as an instance of the appropriate format-specific *Message* subclass unless a custom message factory was specified when the *Mailbox* instance was initialized.

popitem()
Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a *KeyError* exception. The message is represented as an instance of the appropriate format-specific *Message* subclass unless a custom message factory was specified when the *Mailbox* instance was initialized.

update(*arg*)
Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using *__setitem__()*. As with *__setitem__()*, each *key* must already correspond to a message in the mailbox or else a *KeyError* exception will be raised, so in general it is incorrect for *arg* to be a *Mailbox* instance.

참고: Unlike with dictionaries, keyword arguments are not supported.

---

flush()
Write any pending changes to the filesystem. For some *Mailbox* subclasses, changes are always written immediately and *flush()* does nothing, but you should still make a habit of calling this method.

lock()
Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An *ExternalClashError* is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()
Release the lock on the mailbox, if any.

close()
Flush the mailbox, unlock it if necessary, and close any open files. For some *Mailbox* subclasses, this method does nothing.

Maildir

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, *MaildirMessage* is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

If *create* is True and the *dirname* path exists, it will be treated as an existing maildir without attempting to verify its directory layout.

It is for historical reasons that *dirname* is named as such rather than *path*.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: tmp, new, and cur. Messages are created momentarily in the tmp subdirectory and then moved to the new subdirectory to finalize delivery. A mail user agent may subsequently move the message to the cur subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if '.' is the first character in its name. Folder names are represented by *Maildir* without the leading '.'. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using '.' to delimit levels, e.g., "Archived.2005.07".

참고: The Maildir specification requires the use of a colon (':') in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point ('!') is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!'
```

The colon attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of *Mailbox* in addition to the following:

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return a *Maildir* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return a *Maildir* instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

clean()

Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some *Mailbox* methods implemented by *Maildir* deserve special remarks:

add(*message*)
__setitem__(*key*, *message*)
update(*arg*)

> **경고:** These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()
All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()
Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()
Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)
Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

더 보기:

maildir man page from qmail The original specification of the format.

Using maildir format Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

maildir man page from Courier Another specification of the format. Describes a common extension for supporting folders.

mbox

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)
A subclass of *Mailbox* for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *mboxMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, *mbox* implements the original format, which is sometimes referred to as *mboxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

Some *Mailbox* methods implemented by *mbox* deserve special remarks:

get_file(*key*)
Using the file after calling `flush()` or `close()` on the *mbox* instance may yield unpredictable results or raise an exception.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

더 보기:

mbox man page from qmail A specification of the format and its variations.

mbox man page from tin Another specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats A history of mbox variations.

MH

class mailbox.**MH**(*path*, *factory=None*, *create=True*)
A subclass of *Mailbox* for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MHMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called `.mh_sequences` in each folder.

The *MH* class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**'s behaviors. In particular, it does not modify and is not affected by the `context` or `.mh_profile` files that are used by **mh** to store its state and configuration.

MH instances have all of the methods of *Mailbox* in addition to the following:

list_folders()
Return a list of the names of all folders.

get_folder(*folder*)
Return an *MH* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)
Create a folder whose name is *folder* and return an *MH* instance representing it.

remove_folder(*folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

get_sequences()
Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(*sequences*)
Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by *get_sequences()*.

pack()
Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

참고: Already-issued keys are invalidated by this operation and should not be subsequently used.

Some *Mailbox* methods implemented by *MH* deserve special remarks:

remove(*key*)
__delitem__(*key*)
discard(*key*)
These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)
Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()
All changes to MH mailboxes are immediately applied, so this method does nothing.

close()
MH instances do not keep any open files, so this method is equivalent to *unlock()*.

더 보기:

nmh - Message Handling System Home page of **nmh**, an updated version of the original **mh**.

MH & nmh: Email for Users & Programmers A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

Babyl

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)
A subclass of *Mailbox* for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *BabylMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of *Mailbox* in addition to the following:

get_labels()
Return a list of the names of all user-defined labels used in the mailbox.

참고: The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some *Mailbox* methods implemented by *Babyl* deserve special remarks:

get_file(*key*)
In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an *io.BytesIO* instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

더 보기:

Format of Version 5 Babyl Files A specification of the Babyl format.

Reading Mail with Rmail The Rmail manual, with some information on Babyl semantics.

MMDF

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)
A subclass of *Mailbox* for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MMDFMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are "From ", but additional occurrences of "From " are not transformed to ">From " when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some *Mailbox* methods implemented by *MMDF* deserve special remarks:

get_file(*key*)
Using the file after calling `flush()` or `close()` on the *MMDF* instance may yield unpredictable results or raise an exception.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

더 보기:

mmdf man page from tin A specification of MMDF format from the documentation of tin, a newsreader.

MMDF A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

20.4.2 Message objects

class mailbox.**Message**(*message=None*)

A subclass of the *email.message* module's *Message*. Subclasses of *mailbox.Message* add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an *email.message.Message* instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a *Message* instance. If *message* is a string, a byte string, or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that *Message* instances be used to represent messages retrieved using *Mailbox* instances. In some situations, the time and memory required to generate *Message* representations might not be acceptable. For such situations, *Mailbox* instances also offer string and file-like representations, and a custom message factory may be specified when a *Mailbox* instance is initialized.

MaildirMessage

class mailbox.**MaildirMessage**(*message=None*)

A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in `cur` has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in `new`.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

Flag	Meaning	Explanation
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

MaildirMessage instances offer the following methods:

get_subdir()

Return either "new" (if the message should be stored in the `new` subdirectory) or "cur" (if the message should be stored in the `cur` subdirectory).

> **참고:** A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message is has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

set_subdir(*subdir*)
Set the subdirectory the message should be stored in. Parameter *subdir* must be either "new" or "cur".

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if "info" contains experimental semantics.

set_flags(*flags*)
Set the flags specified by *flags* and unset all others.

add_flag(*flag*)
Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current "info" is overwritten whether or not it contains experimental information rather than flags.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If "info" contains experimental information rather than flags, the current "info" is not modified.

get_date()
Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(*date*)
Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info()
Return a string containing the "info" for a message. This is useful for accessing and modifying "info" that is experimental (i.e., not a list of flags).

set_info(*info*)
Set "info" to *info*, which should be a string.

When a *MaildirMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
"cur" subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a *MaildirMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
"cur" subdirectory	"unseen" sequence
"cur" subdirectory and S flag	no "unseen" sequence
F flag	"flagged" sequence
R flag	"replied" sequence

When a *MaildirMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
“cur” subdirectory	“unseen” label
“cur” subdirectory and S flag	no “unseen” label
P flag	“forwarded” or “resent” label
R flag	“answered” label
T flag	“deleted” label

mboxMessage

class mailbox.**mboxMessage**(*message=None*)

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From ” that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

get_from()

Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the “From ” line to *from_*, which should be specified without a leading “From ” or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or `True` (to use *time.gmtime()*).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an *mboxMessage* instance is created based upon a *MaildirMessage* instance, a "From " line is generated based upon the *MaildirMessage* instance's delivery date, and the following conversions take place:

Resulting state	*MaildirMessage* state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an *mboxMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
R flag and O flag	no "unseen" sequence
O flag	"unseen" sequence
F flag	"flagged" sequence
A flag	"replied" sequence

When an *mboxMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
R flag and O flag	no "unseen" label
O flag	"unseen" label
D flag	"deleted" label
A flag	"answered" label

When a *Message* instance is created based upon an *MMDFMessage* instance, the "From " line is copied and all flags directly correspond:

Resulting state	*MMDFMessage* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

MHMessage

class mailbox.**MHMessage**(*message=None*)
A message with MH-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much the same way flags are used with other formats, as follows:

Sequence	Explanation
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

MHMessage instances offer the following methods:

get_sequences()
Return a list of the names of sequences that include this message.

set_sequences(*sequences*)
Set the list of sequences that include this message.

add_sequence(*sequence*)
Add *sequence* to the list of sequences that include this message.

remove_sequence(*sequence*)
Remove *sequence* from the list of sequences that include this message.

When an *MHMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

Resulting state	*MaildirMessage* state
"unseen" sequence	no S flag
"replied" sequence	R flag
"flagged" sequence	F flag

When an *MHMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
"unseen" sequence	no R flag
"replied" sequence	A flag
"flagged" sequence	F flag

When an *MHMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
"unseen" sequence	"unseen" label
"replied" sequence	"answered" label

BabylMessage

class mailbox.**BabylMessage**(*message=None*)
A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

Label	Explanation
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The *BabylMessage* class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

BabylMessage instances offer the following methods:

get_labels()
Return a list of labels on the message.

set_labels(*labels*)
Set the list of labels on the message to *labels*.

add_label(*label*)
Add *label* to the list of labels on the message.

remove_label(*label*)
Remove *label* from the list of labels on the message.

get_visible()
Return an *Message* instance whose headers are the message's visible headers and whose body is empty.

set_visible(*visible*)
Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode).

update_visible()
When a *BabylMessage* instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date*, *From*, *Reply-To*, *To*, *CC*, and *Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a *BabylMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

Resulting state	*MaildirMessage* state
"unseen" label	no S flag
"deleted" label	T flag
"answered" label	R flag
"forwarded" label	P flag

When a *BabylMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
"unseen" label	no R flag
"deleted" label	D flag
"answered" label	A flag

When a *BabylMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
“unseen” label	“unseen” sequence
“answered” label	“replied” sequence

MMDFMessage

class mailbox.**MMDFMessage**(*message=None*)

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with “From “. Likewise, flags that indicate the state of the message are typically stored in *Status* and *X-Status* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

MMDFMessage instances offer the following methods, which are identical to those offered by *mboxMessage*:

get_from()

Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the “From ” line to *from_*, which should be specified without a leading “From ” or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or `True` (to use *time.gmtime()*).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an *MMDFMessage* instance is created based upon a *MaildirMessage* instance, a "From " line is generated based upon the *MaildirMessage* instance's delivery date, and the following conversions take place:

Resulting state	*MaildirMessage* state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an *MMDFMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
R flag and O flag	no "unseen" sequence
O flag	"unseen" sequence
F flag	"flagged" sequence
A flag	"replied" sequence

When an *MMDFMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
R flag and O flag	no "unseen" label
O flag	"unseen" label
D flag	"deleted" label
A flag	"answered" label

When an *MMDFMessage* instance is created based upon an *mboxMessage* instance, the "From " line is copied and all flags directly correspond:

Resulting state	*mboxMessage* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

20.4.3 Exceptions

The following exception classes are defined in the *mailbox* module:

exception mailbox.**Error**
The based class for all other module-specific exceptions.

exception mailbox.**NoSuchMailboxError**
Raised when a mailbox is expected but is not found, such as when instantiating a *Mailbox* subclass with a path that does not exist (and with the *create* parameter set to `False`), or when opening a folder that does not exist.

exception mailbox.**NotEmptyError**
Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception mailbox.**ExternalClashError**
Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

exception mailbox.**FormatError**
Raised when the data in a file cannot be parsed, such as when an *MH* instance attempts to read a corrupted `.mh_sequences` file.

20.4.4 Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
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                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

20.5 mimetypes — 파일명을 MIME 유형에 매핑

소스 코드: Lib/mimetypes.py

---

mimetypes 모듈은 파일명이나 URL과 파일명 확장자와 연관된 MIME 유형 간의 변환을 제공합니다. 변환은 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*)

Guess the type of a file based on its filename or URL, given by *url*. The return value is a tuple `(type, encoding)` where *type* is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME *content-type* header.

*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.

선택적 *strict* 인자는 알려진 MIME 유형 목록을 IANA에 등록된 공식 유형으로만 제한할지를 지정하는 플래그입니다. *strict*가 `True`(기본값)이면 IANA 유형만 지원됩니다; *strict*가 `False`일 때, 추가로 표준이 아니지만, 일반적으로 사용되는 MIME 유형도 인식됩니다.

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()*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 제공하

는 문자열인데, 선행 점('.')을 포함합니다. 확장자는 특정 데이터 스트림과 연관되었음이 보장되지는 않지만, *guess_type()*에 의해 MIME 유형 *type*으로 매핑됩니다. *type*에 대해 추측할 수 있는 확장이 없으면, None이 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.

If *files* is None the internal data structure is completely rebuilt to its initial default value. This is a stable operation and will produce the same results when called multiple times.

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 None이 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 True로 설정됩니다.

mimetypes.**knownfiles**
일반적으로 설치된 유형 맵 파일 이름의 리스트입니다. 이 파일들은 일반적으로 mime.types로 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.gz에 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']
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

20.5.1 MimeTypes 객체

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.gz에 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_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 True이면, 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: 윈도우.

*strict*가 `True`이면, 정보는 표준 유형 리스트에 추가되고, 그렇지 않으면 비표준 유형 리스트에 추가됩니다.

버전 3.2에 추가.

20.6 base64 — Base16, Base32, Base64, Base85 데이터 인코딩

소스 코드: Lib/base64.py

이 모듈은 바이너리 데이터를 인쇄 가능한 ASCII 문자로 인코딩하고 이러한 인코딩을 다시 바이너리 데이터로 디코딩하는 함수를 제공합니다. Base16, Base32 및 Base64 알고리즘을 정의하는 **RFC 3548**에 지정된 인코딩과 사실상의 표준인 Ascii85와 Base85 인코딩에 대한 인코딩과 디코딩 함수를 제공합니다.

RFC 3548 인코딩은 전자 우편으로 안전하게 보내거나, URL의 일부로 사용하거나, HTTP POST 요청의 일부로 포함 할 수 있도록 바이너리 데이터를 인코딩하는 데 적합합니다. 인코딩 알고리즘은 **uuencode** 프로그램과 다릅니다.

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 3548**에 정의된 두 가지(일반, 그리고 URL과 파일 시스템에서 안전한) base-64 알파벳이 모두 지원됩니다.

레거시 인터페이스는 문자열로부터의 디코딩을 지원하지 않지만, 파일 객체에서 인코딩과 디코딩하는 함수를 제공합니다. Base64 표준 알파벳만 지원하며, **RFC 2045**에 따라 76자마다 개행 문자를 추가합니다. **RFC 2045** 지원을 원한다면 아마도 대신 *email* 패키지를 보고 싶을 것입니다.

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.

최신 인터페이스는 다음과 같은 것들을 제공합니다:

base64.**b64encode**(*s*, *altchars=None*)
Base64를 사용하여 바이트열류 객체 *s*를 인코딩하고 인코딩된 *bytes*를 반환합니다.

선택적 *altchars*는 +와 / 문자의 대체 알파벳을 지정하는 최소 길이 2(추가 문자는 무시됩니다)의 바이트열류 객체여야 합니다. 이를 통해 응용 프로그램은 URL이나 파일 시스템에서 안전한 Base64 문자열을 생성할 수 있습니다. 기본값은 `None`이며, 표준 Base64 알파벳이 사용됩니다.

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 최소 길이 2(추가 문자는 무시됩니다)의 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.

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 3548은 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.**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.

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 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.

버전 3.4에 추가.

base64.**a85decode**(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\v'*)
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.

버전 3.4에 추가.

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.

버전 3.4에 추가.

base64.**b85decode**(*b*)
base85로 인코딩된 바이트열류 객체나 ASCII 문자열 *b*를 디코딩하고 디코딩된 *bytes*를 반환합니다. 필요하면, 채우기는 묵시적으로 제거됩니다.

버전 3.4에 추가.

레거시 인터페이스:

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.

버전 3.1에 추가.

base64.**decodestring**(*s*)
*decodebytes()*의 폐지된 별칭.

버전 3.1부터 폐지.

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.

버전 3.1에 추가.

base64.**encodestring**(*s*)
*encodebytes()*의 폐지된 별칭.

버전 3.1부터 폐지.

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'
```

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 o
5.2 절, "Base64 Content-Transfer-Encoding" 은 base64 인코딩의 정의를 제공합니다.

20.7 binhex — binhex4 파일 인코딩과 디코딩

소스 코드: Lib/binhex.py

이 모듈은 binhex4 형식의 파일을 인코드하고 디코드합니다. ASCII 형식의 매킨토시 파일 표현을 허용하는 형식입니다. 데이터 포크만 처리됩니다.

binhex 모듈은 다음 함수를 정의합니다:

binhex.**binhex**(*input*, *output*)
파일명 *input*을 가진 바이너리 파일을 binhex 파일 *output*으로 변환합니다. *output* 매개 변수는 파일명이나 파일류 객체(`write()`와 `close()` 메서드를 지원하는 임의의 객체)일 수 있습니다.

binhex.**hexbin**(*input*, *output*)
binhex 파일 *input*을 디코드합니다. *input*은 파일명이나 `read()` 와 `close()` 메서드를 지원하는 파일류 객체일 수 있습니다. *output* 인자가 `None`이 아니면 결과 파일은 그 이름의 파일에 기록됩니다. *output*이 `None`이면 출력 파일명은 binhex 파일에서 읽습니다.

다음 예외도 정의됩니다:

exception binhex.**Error**
binhex 형식을 사용하여 무언가를 인코드할 수 없거나(예를 들어, 파일명이 파일명 필드에 들어가기에는 너무 길어서), 입력이 제대로 인코드 된 binhex 데이터가 아닐 때 발생하는 예외.

더 보기:

모듈 *binascii* ASCII에서 바이너리로, 바이너리에서 ASCII로의 변환이 포함된 지원 모듈.

20.7.1 노트

코더 및 디코더에 대한 대안적인 더 강력한 인터페이스가 있습니다, 자세한 내용은 소스를 참조하세요.

매킨토시 이외의 플랫폼에서 텍스트 파일을 코딩하거나 디코딩하면, 오래된 매킨토시 개행 규칙(줄의 끝으로 캐리지 리턴사용)을 계속 사용하게 됩니다.

20.8 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 *uu*, *base64* 또는 *binhex*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*)
: base64 데이터 블록을 바이너리로 역변환하고 바이너리 데이터를 반환합니다. 한 번에 한 줄 이상을 전달할 수 있습니다.

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.**a2b_hqx**(*string*)
: RLE 압축 해제 없이 binhex4 형식의 ASCII data를 바이너리로 변환합니다. 이 문자열에는 완전한 바이너리 바이트가 포함되거나, (binhex4 데이터의 마지막 부분에서) 나머지 비트가 0이어야 합니다.

binascii.**rledecode_hqx**(*data*)
: binhex4 표준에 따라, data에 대해 RLE 압축 해제를 수행합니다. 이 알고리즘은 바이트 다음에 오는 0x90을 반복 표시기로 사용하고, 그 뒤에 카운트가 옵니다. 카운트 0은 바이트 값 0x90을 지정합니다. 이 루틴은 data 입력 데이터가 불완전한 반복 표시기로 끝나지 않는 한(이때는 *Incomplete* 예외가 발생합니다) 압축 해제된 데이터를 반환합니다.

버전 3.2에서 변경: 바이트열이나 bytearray 객체만 입력으로 허용합니다.

binascii.**rlecode_hqx**(*data*)
binhex4 스타일의 RLE 압축을 *data*에 대해 수행하고 결과를 반환합니다.

binascii.**b2a_hqx**(*data*)
hexbin4 바이너리에서 ASCII로의 변환을 수행하고 결과 문자열을 반환합니다. 인자는 이미 RLE로 코드화되어 있어야 하며, (마지막 조각을 제외하고) 길이가 3의 배수여야 합니다.

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 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"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

버전 3.0에서 변경: 결과는 항상 부호 없는 정수입니다. 모든 파이썬 버전과 플랫폼에서 같은 숫자 값을 생성하려면, `crc32(data) & 0xffffffff`를 사용하십시오.

binascii.**b2a_hex**(*data*)
binascii.**hexlify**(*data*)
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.

binascii.**a2b_hex**(*hexstr*)
binascii.**unhexlify**(*hexstr*)
16진수 문자열 *hexstr*로 표현된 바이너리 데이터를 반환합니다. 이 함수는 *b2a_hex()*의 역함수입니다. *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 ***binhex*** 매킨토시에서 사용되는 binhex 형식 지원.

모듈 ***uu*** 유닉스에서 사용되는 UU 인코딩 지원.

모듈 ***quopri*** MIME 이메일 메시지에 사용되는 quoted-printable 인코딩 지원.

20.9 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

20.10 uu — uuencode 파일 인코딩과 디코딩

소스 코드: Lib/uu.py

이 모듈은 uuencode 형식으로 파일을 인코딩과 디코딩해서, 임의의 바이너리 데이터를 ASCII 전용 연결을 통해 전송할 수 있도록 합니다. 파일 인자를 기대하는 모든 위치에서 파일류 객체를 사용할 수 있습니다. 이전 버전과의 호환성을 위해, 경로명을 포함하는 문자열도 허용되며 해당 파일은 읽기와 쓰기 용으로 열립니다; 경로명 `'-'`는 표준 입력이나 출력을 의미하는 것으로 이해됩니다. 그러나, 이 인터페이스는 폐지되었습니다; 호출자가 파일을 스스로 여는 것이 더 좋으며, 필요한 경우 윈도우에서 모드가 `'rb'` 나 `'wb'`인지 확인하십시오.

이 코드는 Lance Ellinghouse가 작성했으며, Jack Jansen이 수정했습니다.

uu 모듈은 다음 함수를 정의합니다:

uu.**encode**(*in_file*, *out_file*, *name=None*, *mode=None*, *, *backtick=False*)
in_file 파일을 *out_file* 파일로 uuencode합니다. uuencode 된 파일은 파일을 디코딩한 결과의 기본값으로 *name* 과 *mode*를 지정하는 헤더를 갖습니다. 기본 기본값은 *in_file* 에서 얻거나, 각각 `'-'` 과 `0o666`입니다. *backtick*이 참이면, 0은 스페이스 대신에 `'`'로 표현됩니다.

버전 3.7에서 변경: *backtick* 매개 변수가 추가되었습니다.

uu.**decode**(*in_file*, *out_file=None*, *mode=None*, *quiet=False*)

이 호출은 uuencode 된 파일 *in_file*를 디코딩하여, 파일 *out_file*에 결과를 저장합니다. *out_file*이 경로명이면, 파일을 만들어야 할 때 *mode*를 사용하여 사용 권한 비트를 설정합니다. *out_file* 과 *mode*의 기본값은 uuencode 헤더에서 가져옵니다. 그러나, 헤더에 지정된 파일이 이미 존재하면, *uu.Error*가 발생합니다.

입력이 잘못된 uuencoder에 의해 만들어졌고, 파이썬이 그 에러로부터 복구할 수 있다면, *decode()*는 표준 에러에 경고를 인쇄할 수 있습니다. *quiet*를 참으로 설정하면, 이 경고가 사라집니다.

exception uu.**Error**

*Exception*의 서브 클래스인데, 다양한 상황(가령 위에 언급한 것과 같은, 하지만 형식이 잘못된 헤더나, 잘린 입력 파일도 포함됩니다)에서 *uu.decode()*가 발생시킬 수 있습니다.

더 보기:

모듈 *binascii* ASCII와 바이너리 간의 변환을 포함하는 지원 모듈.

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*)
문자열 *s*의 문자 `&`, < 및 >를 HTML-안전 시퀀스로 변환합니다. HTML에 이러한 문자가 포함될 수 있는 텍스트를 표시해야 할 때 사용하십시오. 선택적 플래그 *quote*가 참이면, 문자 (") 와 (') 도 변환됩니다; `<a href="...">` 에서처럼 따옴표로 구분된 HTML 어트리뷰트에 포함하는 데 도움이 됩니다.

버전 3.2에 추가.

`html.`**`unescape`**(*s*)
문자열 *s*의 모든 이름과 숫자 문자 참조(예를 들어, `&gt;`, `&#62;`, `&#x3e;`)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.

버전 3.4에 추가.

`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 `True`입니다.

21.2.1 HTML 구문 분석기 응용 프로그램 예제

기본 예제로, 다음은 *HTMLParser* 클래스를 사용하여 시작 태그, 종료 태그 및 데이터를 만날 때마다 인쇄하는 간단한 HTML 구문 분석기입니다:

```
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!
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 태그의 시작(예를 들어, `<div id="main">`)을 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 True이면, 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 True이면 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

다음 클래스는 더 많은 예를 설명하는 데 사용할 구문 분석기를 구현합니다:

```
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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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'):
            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()
```

doctype 구문 분석하기:

```
>>> 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"
```

몇 가지 어트리뷰트를 가진 요소와 제목을 구문 분석하기:

```
>>> 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
```

`script`와 `style` 요소의 내용은 더 구문 분석하지 않고 있는 그대로 반환됩니다:

```
>>> 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
```

주석 구문 분석하기:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

이름있는 문자 참조와 숫자 문자 참조를 구문 분석하고 올바른 문자로 변환합니다(참고: 이 3개의 참조는 모두 '>'와 동등합니다):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

불완전한 청크를 *feed()*로 보내는 것이 작동합니다만, *handle_data()*가 두 번 이상 호출될 수 있습니다 (*convert_charrefs*가 True로 설정되지 않은 한):

```
>>> 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
```

잘못된 HTML(예를 들어, 따옴표 처리되지 않은 어트리뷰트)을 구문 분석하는 것도 동작합니다:

```
>>> 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.

  버전 3.3에 추가.

html.entities.**entitydefs**
: XHTML 1.0 엔티티 정의를 ISO Latin-1의 치환 텍스트에 매핑하는 딕셔너리.

[1] https://www.w3.org/TR/html5/syntax.html#named-character-references 를 참조하십시오.

html.entities.**name2codepoint**
HTML 엔티티 이름을 유니코드 코드 포인트에 매핑하는 딕셔너리.

html.entities.**codepoint2name**
유니코드 코드 포인트를 HTML 엔티티 이름에 매핑하는 딕셔너리.

21.4 XML 처리 모듈

소스 코드: Lib/xml/

XML 처리를 위한 파이썬의 인터페이스는 xml 패키지로 묶여있습니다.

> **경고:** XML 모듈은 잘못되었거나 악의적으로 구성된 데이터로부터 안전하지 않습니다. 신뢰할 수 없거나 인증되지 않은 데이터를 구문 분석해야 하면 *XML 취약점*과 *defusedxml와 defusedexpat 패키지* 절을 참조하십시오.

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 취약점

XML 처리 모듈은 악의적으로 구성된 데이터로부터 안전하지 않습니다. 공격자는 XML 기능을 악용하여 서비스 거부 공격을 수행하거나, 로컬 파일에 액세스하거나, 다른 기계로 네트워크 연결을 만들거나, 방화벽을 우회할 수 있습니다.

다음 표는 알려진 공격의 개요와 다양한 모듈이 취약한지를 보여줍니다.

종류	sax	etree	minidom	pulldom	xmlrpc
billion laughs(억만 웃음)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
quadratic blowup(이차 폭발)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
external entity expansion(외부 엔티티 확장)	Safe (5)	Safe (2)	Safe (3)	Safe (5)	안전 (4)
DTD retrieval(DTD 조회)	Safe (5)	안전	안전	Safe (5)	안전
decompression bomb(압축해제 폭탄)	안전	안전	안전	안전	**취약**

1. Expat 2.4.1 and newer is not vulnerable to the "billion laughs" and "quadratic blowup" vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.
2. *xml.etree.ElementTree*는 외부 엔티티를 확장하지 않고 엔티티가 있으면 `ParserError`를 발생시킵니다.
3. *xml.dom.minidom*은 외부 엔티티를 확장하지 않고 확장되지 않은 엔티티를 그대로 반환합니다.
4. `xmlrpclib`는 외부 엔티티를 확장하지 않고 생략합니다.
5. 파이썬 3.7.1부터, 외부 일반 엔티티는 더는 기본적으로 처리되지 않습니다.

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.

external entity expansion(외부 엔티티 확장) 엔티티 선언은 대체 텍스트 이상의 것을 포함할 수 있습니다. 외부 자원이나 지역 파일을 가리킬 수도 있습니다. XML 구문 분석기는 자원에 액세스하고 그 내용을 XML 문서에 포함합니다.

DTD retrieval(DTD 조회) 파이썬의 *xml.dom.pulldom* 같은 일부 XML 라이브러리는 원격이나 지역 위치에서 문서 유형 정의(DTD)를 조회합니다. 이 기능은 외부 엔티티 확장 문제와 비슷한 결과를 줍니다.

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.

PyPI의 defusedxml 설명서에는 모든 알려진 공격 벡터에 대한 추가 정보가 예제와 레퍼런스와 함께 제공됩니다.

21.4.2 `defusedxml`와 `defusedexpat` 패키지

defusedxml은 순수한 파이썬 패키지인데, 모든 악의적인 조작을 방지하도록 수정된, 모든 표준 라이브러리 XML 구문 분석기의 서브 클래스를 제공합니다. 신뢰할 수 없는 XML 데이터를 구문 분석하는 서버 코드에는 이 패키지를 사용하는 것이 좋습니다. 이 패키지에는 XPath 주입과 같은 XML 공격(exploit)에 대한 예제 공격과 확장된 설명서가 함께 제공됩니다.

defusedexpat은 엔티티 확장 DoS 공격에 대한 대책이 있는 수정된 libexpat 와 패치된 `pyexpat` 모듈을 제공합니다. `defusedexpat` 모듈은 여전히 합리적이고 구성 가능한 양의 엔티티 확장을 허용합니다. 이 수정이 향후의 파이썬 배포에 포함될 수 있지만, 이전 버전과의 호환성을 없애기 때문에 파이썬의 버그 수정 배포에는 포함되지 않을 것입니다.

21.5 `xml.etree.ElementTree` — The ElementTree XML API

Source code: Lib/xml/etree/ElementTree.py

The *xml.etree.ElementTree* module implements a simple and efficient API for parsing and creating XML data.

버전 3.3에서 변경: This module will use a fast implementation whenever available. The `xml.etree.cElementTree` module is deprecated.

경고: The *xml.etree.ElementTree* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 취약점*.

21.5.1 Tutorial

This is a short tutorial for using *xml.etree.ElementTree* (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - *ElementTree* represents the whole XML document as a tree, and *Element* represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the *ElementTree* level. Interactions with a single XML element and its sub-elements are done on the *Element* level.

Parsing XML

We'll be using the following XML document as the sample data for this section:

```
<?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>
```

We can import this data by reading from a file:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly from a string:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() parses XML from a string directly into an *Element*, which is the root element of the parsed tree. Other parsing functions may create an *ElementTree*. Check the documentation to be sure.

As an *Element*, `root` has a tag and a dictionary of attributes:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```
>>> root[0][1].text
'2008'
```

참고: Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom *TreeBuilder* instance to the *XMLParser* constructor.

Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an *XMLParser* and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed *Element* objects.

The most powerful tool for doing this is *XMLPullParser*. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with *XMLPullParser.feed()* calls. To get the parsed XML elements, call *XMLPullParser.read_events()*. Here is an example:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, *XMLPullParser* can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at *iterparse()*. It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

Finding interesting elements

Element has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, *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() finds only elements with a tag which are direct children of the current element. *Element.find()* finds the *first* child with a particular tag, and *Element.text* accesses the element's text content. *Element.get()* accesses the element's attributes:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using *XPath*.

Modifying an XML File

ElementTree provides a simple way to build XML documents and write them to files. The *ElementTree.write()* method serves this purpose.

Once created, an *Element* object may be manipulated by directly changing its fields (such as *Element.text*), adding and modifying attributes (*Element.set()* method), as well as adding new children (for example with *Element.append()*).

Let's say we want to add one to each country's rank, and add an `updated` attribute to the rank element:

```
>>> 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')
```

Our XML now looks like this:

```
<?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>
```

We can remove elements using *Element.remove()*. Let's say we want to remove all countries with a rank higher than 50:

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?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>
</data>
```

Building XML documents

The *SubElement()* function also provides a convenient way to create new sub-elements for a given element:

```
>>> 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>
```

Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full *URI*. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix "fictional" and the other serving as the default namespace:

```
<?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>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a *find()* or *findall()*:

```
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)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
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)
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

Additional resources

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs.

21.5.2 XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

Example

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the `countrydata` XML document from the *Parsing XML* section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

Supported XPath syntax

Syntax	Meaning
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`.
`*`	Selects all child elements. For example, `*/egg` selects all grandchildren named `egg`.
`.`	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//egg` selects all `egg` elements in the entire tree.
`..`	Selects the parent element. Returns `None` if the path attempts to reach the ancestors of the start element (the element `find` was called on).
`[@attrib]`	Selects all elements that have the given attribute.
`[@attrib='value']`	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
`[tag]`	Selects all elements that have a child named `tag`. Only immediate children are supported.
`[.='text']`	Selects all elements whose complete text content, including descendants, equals the given `text`. 버전 3.7에 추가.
`[tag='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, equals the given `text`.
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression `last()` (for the last position), or a position relative to the last position (e.g. `last()-1`).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

21.5.3 Reference

Functions

`xml.etree.ElementTree.`**`Comment`**(*text=None*)
Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

Note that *XMLParser* skips over comments in the input instead of creating comment objects for them. An *ElementTree* will only contain comment nodes if they have been inserted into to the tree using one of the *Element* methods.

`xml.etree.ElementTree.`**`dump`**(*elem*)
Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

`xml.etree.ElementTree.`**`fromstring`**(*text*, *parser=None*)
Parses an XML section from a string constant. Same as *XML()*. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)
Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

버전 3.2에 추가.

xml.etree.ElementTree.**iselement**(*element*)
Check if an object appears to be a valid element object. *element* is an element instance. Return `True` if this is an element object.

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)
Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or *file object* containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. *parser* must be a subclass of *XMLParser* and can only use the default *TreeBuilder* as a target. Returns an *iterator* providing `(event, elem)` pairs.

Note that while *iterparse()* builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see *XMLPullParser*.

참고: *iterparse()* only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for "end" events instead.

버전 3.4부터 폐지: The *parser* argument.

xml.etree.ElementTree.**parse**(*source*, *parser=None*)
Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *ElementTree* instance.

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)
PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that *XMLParser* skips over processing instructions in the input instead of creating comment objects for them. An *ElementTree* will only contain processing instruction nodes if they have been inserted into to the tree using one of the *Element* methods.

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)
Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

버전 3.2에 추가.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)
Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

xml.etree.ElementTree.**tostring**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns an (optionally) encoded string containing the XML data.

버전 3.4에 추가: The *short_empty_elements* parameter.

xml.etree.ElementTree.**tostringlist**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

버전 3.2에 추가.

버전 3.4에 추가: The *short_empty_elements* parameter.

xml.etree.ElementTree.**XML**(*text*, *parser=None*)

Parses an XML section from a string constant. This function can be used to embed "XML literals" in Python code. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)

Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

21.5.4 XInclude support

This module provides limited support for XInclude directives, via the `xml.etree.ElementInclude` helper module. This module can be used to insert subtrees and text strings into element trees, based on information in the tree.

Example

Here's an example that demonstrates use of the XInclude module. To include an XML document in the current document, use the `{http://www.w3.org/2001/XInclude}include` element and set the **parse** attribute to `"xml"`, and use the **href** attribute to specify the document to include.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

By default, the **href** attribute is treated as a file name. You can use custom loaders to override this behaviour. Also note that the standard helper does not support XPointer syntax.

To process this file, load it as usual, and pass the root element to the `xml.etree.ElementTree` module:

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

The ElementInclude module replaces the `{http://www.w3.org/2001/XInclude}include` element with the root element from the **source.xml** document. The result might look something like this:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

If the **parse** attribute is omitted, it defaults to "xml". The href attribute is required.

To include a text document, use the `{http://www.w3.org/2001/XInclude}include` element, and set the **parse** attribute to "text":

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

The result might look something like:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

21.5.5 Reference

Functions

`xml.etree.ElementInclude.`**`default_loader`**(*href*, *parse*, *encoding=None*)

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either "xml" or "text". *encoding* is an optional text encoding. If not given, encoding is `utf-8`. Returns the expanded resource. If the parse mode is `"xml"`, this is an ElementTree instance. If the parse mode is "text", this is a Unicode string. If the loader fails, it can return None or raise an exception.

`xml.etree.ElementInclude.`**`include`**(*elem*, *loader=None*)

This function expands XInclude directives. *elem* is the root element. *loader* is an optional resource loader. If omitted, it defaults to *`default_loader()`*. If given, it should be a callable that implements the same interface as *`default_loader()`*. Returns the expanded resource. If the parse mode is `"xml"`, this is an ElementTree instance. If the parse mode is "text", this is a Unicode string. If the loader fails, it can return None or raise an exception.

Element Objects

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)
Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

tag
A string identifying what kind of data this element represents (the element type, in other words).

text
tail
These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the *text* attribute holds either the text between the element's start tag and its first child or end tag, or `None`, and the *tail* attribute holds either the text between the element's end tag and the next tag, or `None`. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the *a* element has `None` for both *text* and *tail* attributes, the *b* element has *text* `"1"` and *tail* `"4"`, the *c* element has *text* `"2"` and *tail* `None`, and the *d* element has *text* `None` and *tail* `"3"`.

To collect the inner text of an element, see *itertext()*, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

attrib
A dictionary containing the element's attributes. Note that while the *attrib* value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()
Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

get(*key*, *default=None*)
Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()
Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()
Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(*key*, *value*)
Set the attribute *key* on the element to *value*.

The following methods work on the element's children (subelements).

append(*subelement*)
Adds the element *subelement* to the end of this element's internal list of subelements. Raises *TypeError* if *subelement* is not an *Element*.

extend(*subelements*)
Appends *subelements* from a sequence object with zero or more elements. Raises *TypeError* if a subelement is not an *Element*.

버전 3.2에 추가.

find(*match*, *namespaces=None*)
Finds the first subelement matching *match*. *match* may be a tag name or a *path*. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name.

findall(*match*, *namespaces=None*)
Finds all matching subelements, by tag name or *path*. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

findtext(*match*, *default=None*, *namespaces=None*)
Finds text for the first subelement matching *match*. *match* may be a tag name or a *path*. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name.

getchildren()
버전 3.2부터 폐지: Use `list(elem)` or iteration.

getiterator(*tag=None*)
버전 3.2부터 폐지: Use method *Element.iter()* instead.

insert(*index*, *subelement*)
Inserts *subelement* at the given position in this element. Raises *TypeError* if *subelement* is not an *Element*.

iter(*tag=None*)
Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

버전 3.2에 추가.

iterfind(*match*, *namespaces=None*)
Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

버전 3.2에 추가.

itertext()
Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

버전 3.2에 추가.

makeelement(*tag*, *attrib*)
Creates a new element object of the same type as this element. Do not call this method, use the *SubElement()* factory function instead.

remove(*subelement*)
Removes *subelement* from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

Element objects also support the following sequence type methods for working with subelements: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution: Elements with no subelements will test as `False`. This behavior will change in future versions. Use specific `len(elem)` or `elem is None` test instead.

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

ElementTree Objects

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

_setroot(*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

find(*match*, *namespaces=None*)

Same as *Element.find()*, starting at the root of the tree.

findall(*match*, *namespaces=None*)

Same as *Element.findall()*, starting at the root of the tree.

findtext(*match*, *default=None*, *namespaces=None*)

Same as *Element.findtext()*, starting at the root of the tree.

getiterator(*tag=None*)

버전 3.2부터 폐지: Use method *ElementTree.iter()* instead.

getroot()

Returns the root element for this tree.

iter(*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

iterfind(*match*, *namespaces=None*)

Same as *Element.iterfind()*, starting at the root of the tree.

버전 3.2에 추가.

parse(*source*, *parser=None*)

Loads an external XML section into this element tree. *source* is a file name or *file object*. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns the section root element.

write(*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*, *, *short_empty_elements=True*)

Writes the element tree to a file, as XML. *file* is a file name, or a *file object* opened for writing. *encoding*[1] is the output encoding (default is US-ASCII). *xml_declaration* controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or Unicode (default is `None`). *default_namespace* sets the default XML namespace (for "xmlns"). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (*str*) or binary (*bytes*). This is controlled by the *encoding* argument. If *encoding* is `"unicode"`, the output is a string; otherwise, it's binary. Note that this may conflict with the type of *file* if it's an open *file object*; make sure you do not try to write a string to a binary stream and vice versa.

버전 3.4에 추가: The *short_empty_elements* parameter.

This is the XML file that is going to be manipulated:

```
<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>
```

Example of changing the attribute "target" of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

QName Objects

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. *QName* instances are opaque.

TreeBuilder Objects

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*)

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. *element_factory*, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

close()

Flushes the builder buffers, and returns the toplevel document element. Returns an *Element* instance.

data(*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

end(*tag*)
Closes the current element. *tag* is the element name. Returns the closed element.

start(*tag*, *attrs*)
Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

In addition, a custom *TreeBuilder* object can provide the following method:

doctype(*name*, *pubid*, *system*)
Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default *TreeBuilder* class.

버전 3.2에 추가.

XMLParser Objects

class xml.etree.ElementTree.**XMLParser**(*html=0*, *target=None*, *encoding=None*)
This class is the low-level building block of the module. It uses *xml.parsers.expat* for efficient, event-based parsing of XML. It can be fed XML data incrementally with the *feed()* method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard *TreeBuilder* is used. The *html* argument was historically used for backwards compatibility and is now deprecated. If *encoding*[1] is given, the value overrides the encoding specified in the XML file.

버전 3.4부터 폐지: The *html* argument. The remaining arguments should be passed via keyword to prepare for the removal of the *html* argument.

close()
Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction; by default, this is the toplevel document element.

doctype(*name*, *pubid*, *system*)
버전 3.2부터 폐지: Define the *TreeBuilder.doctype()* method on a custom TreeBuilder target.

feed(*data*)
Feeds data to the parser. *data* is encoded data.

XMLParser.feed() calls *target*'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. *XMLParser.close()* calls *target*'s method `close()`. *XMLParser* can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser Objects

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of *XMLParser*, but instead of pushing calls to a callback target, *XMLPullParser* collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings "start", "end", "start-ns" and "end-ns" (the "ns" events are used to get detailed namespace information). If *events* is omitted, only "end" events are reported.

feed(*data*)

Feed the given bytes data to the parser.

close()

Signal the parser that the data stream is terminated. Unlike *XMLParser.close()*, this method always returns *None*. Any events not yet retrieved when the parser is closed can still be read with *read_events()*.

read_events()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields (event, elem) pairs, where *event* is a string representing the type of event (e.g. "end") and *elem* is the encountered *Element* object.

Events provided in a previous call to *read_events()* will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from *read_events()* will have unpredictable results.

참고: *XMLPullParser* only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for "end" events instead.

버전 3.4에 추가.

Exceptions

class xml.etree.ElementTree.**ParseError**

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

code

A numeric error code from the expat parser. See the documentation of *xml.parsers.expat* for the list of error codes and their meanings.

position

A tuple of *line*, *column* numbers, specifying where the error occurred.

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 `namespaceURI`에서 발견되거나, 이름 공간별 메서드에 대한 *namespaceURI* 매개 변수로 사용됩니다.

xml.dom.**XML_NAMESPACE**
예약된 접두어 `xml`과 연관된 이름 공간 URI, *Namespaces in XML <https://www.w3.org/TR/REC-xml-names/>*에서 정의됩니다(4절).

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.
`ProcessingInstruction`	*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 `DOMImplementation`을 사용하여 새로운 `Document`와 `DocumentType` 객체를 만드는 기능을 추가했습니다.

`DOMImplementation.`**`hasFeature`**(*feature*, *version*)
Return `True` if the feature identified by the pair of strings *feature* and *version* is implemented.

`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 `None`이어야 합니다. 파이썬 DOM API에서, `Element` 자식이 만들어지지 않음을 표시하기 위해 처음 두 인자는 `None`일 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**()
Return True if the node has any attributes.

Node.**hasChildNodes**()
Return True if the node has any child nodes.

Node.**isSameNode**(*other*)
Return True if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

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 DocumentType은 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 None입니다.

DocumentType.**systemId**
: 문서 형 정의의 외부 부분 집합(external subset)에 대한 시스템 식별자. 문자열로 표현된 URI이거나 None입니다.

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 엔터티의 정의를 제공하는 NamedNodeMap입니다. 엔티티 이름이 두 번 이상 정의되면, 첫 번째 정의 만 제공됩니다 (XML 권장 사항에 따라 다른 정의는 무시됩니다). 구문 분석기가 정보를 제공하지 않거나 정의된 엔터티가 없으면 None일 수 있습니다.

DocumentType.**notations**
: 표기법의 정의를 제공하는 NamedNodeMap입니다. 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 None일 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*)
Return `True` if the element has an attribute named by *name*.

Element.**hasAttributeNS**(*namespaceURI*, *localName*)
Return `True` if the element has an attribute named by *namespaceURI* and *localName*.

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 *InuseAttributeErr*가 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 *InuseAttributeErr*가 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 반환값과 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.

경고: *xml.dom.minidom* 모듈은 악의적으로 구성된 데이터로부터 안전하지 않습니다. 신뢰할 수 없거나 인증되지 않은 데이터를 구문 분석해야 하면 *XML 취약점*를 참조하십시오.

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')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

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 `Document`를 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 `Document`를 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:
    ... # Work with dom.
```

Node.**writexml**(*writer*, *indent=""*, *addindent=""*, *newl=""*)
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.

Node.**toxml**(*encoding=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.

Node.**toprettyxml**(*indent="\t", newl="\n", encoding=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.

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)
```

(다음 페이지에 계속)

[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.

(이전 페이지에서 계속)

```
def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

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 DOM입니다.

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.

경고: *xml.dom.pulldom* 모듈은 악의적으로 구성된 데이터로부터 안전하지 않습니다. 신뢰할 수 없거나 인증되지 않은 데이터를 구문 분석해야 하면 *XML 취약점*를 참조하십시오.

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())
```

event는 상수이며 다음 중 하나일 수 있습니다:

- START_ELEMENT
- END_ELEMENT
- COMMENT

- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node는 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*)

getEvent()
*event*와 현재 *node*를 포함하는 튜플을 반환합니다. 노드는 이벤트가 START_DOCUMENT와 같으면 xml.dom.minidom.Document, 이벤트가 START_ELEMENT나 END_ELEMENT와 같으면 xml.dom.minidom.Element, 이벤트가 CHARACTERS와 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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<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.

경고: *xml.sax* 모듈은 악의적으로 구성된 데이터로부터 안전하지 않습니다. 신뢰할 수 없거나 인증되지 않은 데이터를 구문 분석해야 하면 *XML 취약점*을 참조하십시오.

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 문자열 리스트여야 합니다. *parser_list*에 나열된 모듈은 기본 구문 분석기 목록에 있는 모듈보다 먼저 사용됩니다.

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, None이거나 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 — Base classes for SAX handlers

Source code: Lib/xml/sax/handler.py

The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module *xml.sax.handler*, so that all methods get default implementations.

class xml.sax.handler.**ContentHandler**
This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.**DTDHandler**
Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.**EntityResolver**
Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.**ErrorHandler**
Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, *xml.sax.handler* provides symbolic constants for the feature and property names.

xml.sax.handler.**feature_namespaces**

value: "http://xml.org/sax/features/namespaces"
true: Perform Namespace processing.
false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_namespace_prefixes**

value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_string_interning**

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_validation**

value: "http://xml.org/sax/features/validation"

true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_ges**

value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**all_features**
List of all features.

xml.sax.handler.**property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.**property_declaration_handler**

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.**property_dom_node**

value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**property_xml_string**

value: "http://xml.org/sax/properties/xml-string"
data type: String
description: The literal string of characters that was the source for the current event.
access: read-only

xml.sax.handler.**all_properties**
List of all known property names.

21.10.1 ContentHandler Objects

Users are expected to subclass `ContentHandler` to support their application. The following methods are called by the parser on the appropriate events in the input document:

`ContentHandler.`**`setDocumentLocator`**(*locator*)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

`ContentHandler.`**`startDocument`**()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for `setDocumentLocator()`).

`ContentHandler.`**`endDocument`**()
Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

`ContentHandler.`**`startPrefixMapping`**(*prefix*, *uri*)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the `startPrefixMapping()` and `endPrefixMapping()` events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that `startPrefixMapping()` and `endPrefixMapping()` events are not guaranteed to be properly nested relative to each-other: all `startPrefixMapping()` events will occur before the corresponding `startElement()` event, and all `endPrefixMapping()` events will occur after the corresponding `endElement()` event, but their order is not guaranteed.

`ContentHandler.`**`endPrefixMapping`**(*prefix*)
End the scope of a prefix-URI mapping.

See `startPrefixMapping()` for details. This event will always occur after the corresponding `endElement()` event, but the order of `endPrefixMapping()` events is not otherwise guaranteed.

`ContentHandler.`**`startElement`**(*name*, *attrs*)
Signals the start of an element in non-namespace mode.

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 (see *Attributes* 인터페이스) 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*)
Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElement()` event.

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)
Signals the start of an element in namespace mode.

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 (see *AttributesNS 인터페이스*) 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.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

ContentHandler.**endElementNS**(*name*, *qname*)
Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElementNS()` method, likewise the *qname* parameter.

ContentHandler.**characters**(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a string or bytes instance; the `expat` reader module always produces strings.

참고: The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use *content* instead of slicing content with the old *offset* and *length* parameters.

ContentHandler.**ignorableWhitespace**(*whitespace*)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

ContentHandler.**processingInstruction**(*target*, *data*)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

ContentHandler.**skippedEntity**(*name*)
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

21.10.2 DTDHandler Objects

DTDHandler instances provide the following methods:

`DTDHandler.`**`notationDecl`**(*name*, *publicId*, *systemId*)
Handle a notation declaration event.

`DTDHandler.`**`unparsedEntityDecl`**(*name*, *publicId*, *systemId*, *ndata*)
Handle an unparsed entity declaration event.

21.10.3 EntityResolver Objects

`EntityResolver.`**`resolveEntity`**(*publicId*, *systemId*)
Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns *systemId*.

21.10.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the *XMLReader*. If you create an object that implements this interface, then register the object with your *XMLReader*, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a `SAXParseException` as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

`ErrorHandler.`**`error`**(*exception*)
Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

`ErrorHandler.`**`fatalError`**(*exception*)
Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

`ErrorHandler.`**`warning`**(*exception*)
Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

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.

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. '&amp', '<' 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, True로 설정하면 하나의 스스로 닫힌 태그를 출력합니다.

버전 3.2에 추가: *short_empty_elements* 매개 변수.

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*)
Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source – typically a file name or a URL), a file-like object, or an *InputSource* object. When *parse()* returns, the input is completely processed, and the parser object can be discarded or reset.

버전 3.5에서 변경: 문자 스트림 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, SAXNotRecognizedException이 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, SAXNotRecognizedException가 발생합니다. 구문 분석기가 기능이나 해당 설정을 지원하지 않으면 *SAXNotSupportedException*이 발생합니다.

XMLReader.**getProperty**(*propertyname*)
속성 *propertyname*의 현재 설정을 반환합니다. 속성이 인식되지 않으면 SAXNotRecognizedException이 발생합니다. 잘 알려진 속성 이름은 모듈 *xml.sax.handler*에 나열되어 있습니다.

XMLReader.**setProperty**(*propertyname*, *value*)
*propertyname*을 *value*로 설정합니다. 속성이 인식되지 않으면 SAXNotRecognizedException이 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 — Fast XML parsing using Expat

경고: The pyexpat module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 취약점*.

The *xml.parsers.expat* module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object:

exception xml.parsers.expat.**ExpatError**
The exception raised when Expat reports an error. See section *ExpatError Exceptions* for more information on interpreting Expat errors.

exception xml.parsers.expat.**error**
Alias for *ExpatError*.

xml.parsers.expat.**XMLParserType**
The type of the return values from the *ParserCreate()* function.

The *xml.parsers.expat* module contains two functions:

xml.parsers.expat.**ErrorString**(*errno*)
Returns an explanatory string for a given error number *errno*.

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)
Creates and returns a new xmlparser object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding*[1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a *ValueError* will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (' ') and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

StartElementHandler will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the Expat library used by pyexpat, the xmlparser instance returned can only be used to parse a single XML document. Call ParserCreate for each document to provide unique parser instances.

더 보기:

The Expat XML Parser Home page of the Expat project.

21.13.1 XMLParser Objects

xmlparser objects have the following methods:

xmlparser.**Parse**(*data*[, *isfinal*])
Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

xmlparser.**ParseFile**(*file*)
Parse XML data reading from the object *file*. *file* only needs to provide the read(nbytes) method, returning the empty string when there's no more data.

xmlparser.**SetBase**(*base*)
Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the *base* argument to the *ExternalEntityRefHandler()*, *NotationDeclHandler()*, and *UnparsedEntityDeclHandler()* functions.

xmlparser.**GetBase**()
Returns a string containing the base set by a previous call to *SetBase()*, or None if *SetBase()* hasn't been called.

xmlparser.**GetInputContext**()
Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is None.

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])
Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the *ExternalEntityRefHandler()* handler function, described below. The child parser is created with the *ordered_attributes* and *specified_attributes* set to the values of this parser.

xmlparser.**SetParamEntityParsing**(*flag*)
Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE and XML_PARAM_ENTITY_PARSING_ALWAYS. Return true if setting the flag was successful.

xmlparser.**UseForeignDTD**([*flag*])
Calling this with a true value for *flag* (the default) will cause Expat to call the *ExternalEntityRefHandler* with *None* for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the *ExternalEntityRefHandler* will still be called, but the *StartDoctypeDeclHandler* and *EndDoctypeDeclHandler* will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the *Parse()* or *ParseFile()* methods are called; calling it after either of those have been called causes *ExpatError* to be raised with the *code* attribute set to `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`.

`xmlparser` objects have the following attributes:

`xmlparser.`**buffer_size**
The size of the buffer used when *buffer_text* is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

`xmlparser.`**buffer_text**
Setting this to true causes the `xmlparser` object to buffer textual content returned by Expat to avoid multiple calls to the *CharacterDataHandler()* callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

`xmlparser.`**buffer_used**
If *buffer_text* is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when *buffer_text* is false.

`xmlparser.`**ordered_attributes**
Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

`xmlparser.`**specified_attributes**
If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an `xmlparser` object, and will only have correct values once a call to `Parse()` or `ParseFile()` has raised an *xml.parsers.expat.ExpatError* exception.

`xmlparser.`**ErrorByteIndex**
Byte index at which an error occurred.

`xmlparser.`**ErrorCode**
Numeric code specifying the problem. This value can be passed to the *ErrorString()* function, or compared to one of the constants defined in the `errors` object.

`xmlparser.`**ErrorColumnNumber**
Column number at which an error occurred.

`xmlparser.`**ErrorLineNumber**
Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an `xmlparser` object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

`xmlparser.`**CurrentByteIndex**
Current byte index in the parser input.

`xmlparser.`**CurrentColumnNumber**
Current column number in the parser input.

xmlparser.**CurrentLineNumber**
Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an xmlparser object *o*, use o.handlername = func. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)
Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional "standalone" declaration. *version* and *encoding* will be strings, and *standalone* will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)
Called when Expat begins parsing the document type declaration (<!DOCTYPE ...). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or None if omitted. *has_internal_subset* will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.**EndDoctypeDeclHandler**()
Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.**ElementDeclHandler**(*name*, *model*)
Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)
Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are 'CDATA', 'ID', 'IDREF', ... *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

xmlparser.**StartElementHandler**(*name*, *attributes*)
Called for the start of every element. *name* is a string containing the element name, and *attributes* is the element attributes. If *ordered_attributes* is true, this is a list (see *ordered_attributes* for a full description). Otherwise it's a dictionary mapping names to values.

xmlparser.**EndElementHandler**(*name*)
Called for the end of every element.

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)
Called for every processing instruction.

xmlparser.**CharacterDataHandler**(*data*)
Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the *StartCdataSectionHandler*, *EndCdataSectionHandler*, and *ElementDeclHandler* callbacks to collect the required information.

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)
Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use *EntityDeclHandler* instead. (The underlying function in the Expat library has been declared obsolete.)

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)
Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be None for external entities. The *notationName* parameter will be None for

parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)
Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be `None`.

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)
Called when an element contains a namespace declaration. Namespace declarations are processed before the *StartElementHandler* is called for the element on which declarations are placed.

xmlparser.**EndNamespaceDeclHandler**(*prefix*)
Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the *StartNamespaceDeclHandler* was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding *EndElementHandler* for the end of the element.

xmlparser.**CommentHandler**(*data*)
Called for comments. *data* is the text of the comment, excluding the leading `'<!--'` and trailing `'-->'`.

xmlparser.**StartCdataSectionHandler**()
Called at the start of a CDATA section. This and *EndCdataSectionHandler* are needed to be able to identify the syntactical start and end for CDATA sections.

xmlparser.**EndCdataSectionHandler**()
Called at the end of a CDATA section.

xmlparser.**DefaultHandler**(*data*)
Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

xmlparser.**DefaultHandlerExpand**(*data*)
This is the same as the *DefaultHandler()*, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

xmlparser.**NotStandaloneHandler**()
Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)
Called for references to external entities. *base* is the current base, as set by a previous call to *SetBase()*. The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be `None`. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns `0`, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the *DefaultHandler* callback, if provided.

21.13.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

ExpatError.**code**
: Expat's internal error number for the specific error. The *errors.messages* dictionary maps these error numbers to Expat's error messages. For example:

```
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])
```

The *errors* module also provides error message constants and a dictionary *codes* mapping these messages back to the error codes, see below.

ExpatError.**lineno**
: Line number on which the error was detected. The first line is numbered `1`.

ExpatError.**offset**
: Character offset into the line where the error occurred. The first column is numbered `0`.

21.13.3 Example

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
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)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

21.13.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the *xml.parsers.expat.model* module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

xml.parsers.expat.model.**XML_CTYPE_ANY**
: The element named by the model name was declared to have a content model of `ANY`.

xml.parsers.expat.model.**XML_CTYPE_CHOICE**
: The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

xml.parsers.expat.model.**XML_CTYPE_EMPTY**
: Elements which are declared to be `EMPTY` have this model type.

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**
: Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:

xml.parsers.expat.model.**XML_CQUANT_NONE**
: No modifier is given, so it can appear exactly once, as for `A`.

xml.parsers.expat.model.**XML_CQUANT_OPT**
: The model is optional: it can appear once or not at all, as for `A?`.

xml.parsers.expat.model.**XML_CQUANT_PLUS**
: The model must occur one or more times (like `A+`).

xml.parsers.expat.model.**XML_CQUANT_REP**
: The model must occur zero or more times, as for `A*`.

21.13.5 Expat error constants

The following constants are provided in the *xml.parsers.expat.errors* module. These constants are useful in interpreting some of the attributes of the ExpatError exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error *message* and not the numeric error *code*, you do this by comparing its *code* attribute with `errors.codes[errors.XML_ERROR_CONSTANT_NAME]`.

The errors module has the following attributes:

xml.parsers.expat.errors.**codes**
A dictionary mapping numeric error codes to their string descriptions.

버전 3.2에 추가.

xml.parsers.expat.errors.**messages**
A dictionary mapping string descriptions to their error codes.

버전 3.2에 추가.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
A character reference referred to a character which is illegal in XML (for example, character 0, or '&#0;').

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
An attribute was used more than once in a start tag.

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value 0) in a UTF-8 input stream.

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
Something other than whitespace occurred after the document element.

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat was not able to allocate memory internally.

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**
A parameter entity reference was found where it was not allowed.

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**
An incomplete character was found in the input.

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**
An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**
Some unspecified syntax error was encountered.

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**
An end tag did not match the innermost open start tag.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**
Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**
A reference was made to an entity which was not defined.

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**
The document encoding is not supported by Expat.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**
A CDATA marked section was not closed.

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**
The parser determined that the document was not "standalone" though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned `0`.

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**
An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the *xml.parsers.expat* module.

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**
An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**
The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**
A parameter entity contained incomplete markup.

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**
The document contained no document element at all.

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**
There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**
Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**
The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**
An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.errors.**XML_ERROR_ABORTED**
This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_FINISHED**
The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

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.

환경 변수 BROWSER가 있으면, 플랫폼 기본값보다 먼저 시도하기 위해 *os.pathsep*으로 구분된 브라우저 목록으로 해석됩니다. 목록 부분의 값에 문자열 `%s`가 포함되어 있으면, `%s`를 인자 URL로 치환해서 만들어지는 리터럴 브라우저 명령 줄로 해석됩니다; 부분이 `%s`를 포함하지 않으면, 단순히 시작할 브라우저의 이름으로 해석됩니다.[1]

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.

스크립트 **webbrowser**는 모듈의 명령 줄 인터페이스로 사용될 수 있습니다. 인자로 URL을 받아들입니다. 다음과 같은 선택적 매개 변수를 받아들입니다: `-n`은 가능하다면 새 브라우저 창에서 URL을 엽니다; `-t`는 새 브라우저 페이지("탭")에서 URL을 엽니다. 당연히, 이 옵션들은 상호 배타적입니다. 사용 예:

```
python -m webbrowser -t "http://www.python.org"
```

[1] 전체 경로 없이 여기에서 명명된 실행 파일은 PATH 환경 변수에 지정된 디렉터리에서 검색됩니다.

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 True이면, 가능하다면 창을 올립니다(raise) (이것은 많은 창 관리자에서 이 변수의 설정과 관계없이 일어납니다).

일부 플랫폼에서, 이 함수를 사용하여 파일명을 여는 것은 동작하고 운영 체제의 연결된 프로그램이 시작될 수 있습니다. 하지만, 이것은 지원되지도 이식성 있지도 않습니다.

webbrowser.**open_new**(*url*)
가능하다면, 기본 브라우저의 새 창에서 *url*을 엽니다, 그렇지 않으면, 유일한 브라우저 창에서 *url*을 엽니다.

webbrowser.**open_new_tab**(*url*)
가능하다면, 기본 브라우저의 새 페이지("탭")에서 *url*을 엽니다, 그렇지 않으면 *open_new()*와 동등합니다.

webbrowser.**get**(*using=None*)
브라우저 유형 *using*에 대한 제어기 객체를 반환합니다. *using*이 None이면, 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, None이면, 필요할 때 *constructor*가 매개 변수 없이 호출되어 인스턴스를 만듭니다. *instance*가 제공되면, *constructor*는 절대로 호출되지 않으며, None일 수 있습니다.

*preferred*를 True로 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')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSX('default')	(3)
'safari'	MacOSX('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	

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 X 플랫폼에서만.

버전 3.3에 추가: Chrome/Chromium에 대한 지원이 추가되었습니다.

여기 몇 가지 간단한 예제가 있습니다:

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

22.1.1 브라우저 제어기 객체

브라우저 제어기는 세 가지 모듈 수준의 편리 함수와 평행하게 다음과 같은 메서드를 제공합니다:

`controller.`**`open`**(*url*, *new=0*, *autoraise=True*)
이 제어기가 처리하는 브라우저를 사용하여 *url*을 표시합니다. *new*가 1 이면, 가능하다면 새 브라우저 창이 열립니다. *new*가 2 이면, 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 `cgi` — Common Gateway Interface support

Source code: Lib/cgi.py

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

22.2.1 Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server's special `cgi-bin` directory. The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.

The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the "query string" part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```
print("Content-Type: text/html")    # HTML is following
print()                             # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML:

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

22.2.2 Using the cgi module

Begin by writing `import cgi`.

When you write a new script, consider adding these lines:

```
import cgitb
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

It's very helpful to use this feature during script development. The reports produced by *cgitb* provide information that can save you a lot of time in tracking down bugs. You can always remove the `cgitb` line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, use the `FieldStorage` class. If the form contains non-ASCII characters, use the *encoding* keyword parameter set to the value of the encoding defined for the document. It is usually contained in the META tag in the HEAD section of the HTML document or by the *Content-Type* header). This reads the form contents from the standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The `FieldStorage` instance can be indexed like a Python dictionary. It allows membership testing with the `in` operator, and also supports the standard dictionary method *keys()* and the built-in function *len()*. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional *keep_blank_values* keyword parameter when creating the `FieldStorage` instance.

For instance, the following code (which assumes that the *Content-Type* header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the *getlist()* method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as bytes. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data from the `file` attribute before it is automatically closed as part of the garbage collection of the `FieldStorage` instance (the *`read()`* and *`readline()`* methods will return bytes):

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

`FieldStorage` objects also support being used in a `with` statement, which will automatically close them when done.

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the `done` attribute of the object for the field will be set to the value -1.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive *`multipart/*`* encoding). When this occurs, the item will be a dictionary-like `FieldStorage` item. This can be determined by testing its `type` attribute, which should be *`multipart/form-data`* (or perhaps another MIME type matching *`multipart/*`*). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the "old" format (as the query string or as a single data part of type *`application/x-www-form-urlencoded`*), the items will actually be instances of the class `MiniFieldStorage`. In this case, the `list`, `file`, and `filename` attributes are always `None`.

A form submitted via POST that also has a query string will contain both `FieldStorage` and `MiniFieldStorage` items.

버전 3.4에서 변경: The `file` attribute is automatically closed upon the garbage collection of the creating `FieldStorage` instance.

버전 3.5에서 변경: Added support for the context management protocol to the `FieldStorage` class.

22.2.3 Higher Level Interface

The previous section explains how to read CGI form data using the `FieldStorage` class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another `user=foo` pair to the query string, then the script would crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the *upper()* method on a list is not valid (since lists do not have a method of this name) and results in an *AttributeError* exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods *getfirst()* and *getlist()* provided by this higher level interface.

`FieldStorage.`**`getfirst`**(*name*, *default=None*)
: This method always returns only one value associated with form field *name*. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.[1] If no such form field or value exists then the method returns the value specified by the optional parameter *default*. This parameter defaults to `None` if not specified.

`FieldStorage.`**`getlist`**(*name*)
: This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for *name*. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

22.2.4 Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

`cgi.`**`parse`**(*fp=None*, *environ=os.environ*, *keep_blank_values=False*, *strict_parsing=False*, *separator="&"*)
: Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The *keep_blank_values*, *strict_parsing* and *separator* parameters are passed to *urllib.parse.parse_qs()* unchanged.

`cgi.`**`parse_qs`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*)
: This function is deprecated in this module. Use *urllib.parse.parse_qs()* instead. It is maintained here only for backward compatibility.

`cgi.`**`parse_qsl`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*)
: This function is deprecated in this module. Use *urllib.parse.parse_qsl()* instead. It is maintained here only for backward compatibility.

[1] Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.

cgi.**parse_multipart**(*fp*, *pdict*, *encoding="utf-8"*, *errors="replace"*, *separator="&"*)
Parse input of type *multipart/form-data* (for file uploads). Arguments are *fp* for the input file, *pdict* for a dictionary containing other parameters in the *Content-Type* header, and *encoding*, the request encoding.

Returns a dictionary just like *urllib.parse.parse_qs()*: keys are the field names, each value is a list of values for that field. For non-file fields, the value is a list of strings.

This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the `FieldStorage` class instead which is much more flexible.

버전 3.7에서 변경: Added the *encoding* and *errors* parameters. For non-file fields, the value is now a list of strings, not bytes.

버전 3.7.10에서 변경: Added the *separator* parameter.

cgi.**parse_header**(*string*)
Parse a MIME header (such as *Content-Type*) into a main value and a dictionary of parameters.

cgi.**test**()
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

cgi.**print_environ**()
Format the shell environment in HTML.

cgi.**print_form**(*form*)
Format a form in HTML.

cgi.**print_directory**()
Format the current directory in HTML.

cgi.**print_environ_usage**()
Print a list of useful (used by CGI) environment variables in HTML.

cgi.**escape**(*s*, *quote=False*)
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 quotation mark character (`"`) is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in `<a href="...">`. Note that single quotes are never translated.

버전 3.2부터 폐지: This function is unsafe because *quote* is false by default, and therefore deprecated. Use *html.escape()* instead.

22.2.5 Caring about security

There's one important rule: if you invoke an external program (via the *os.system()* or *os.popen()* functions. or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

22.2.6 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory `cgi-bin` in the server tree.

Make sure that your script is readable and executable by "others"; the Unix file mode should be `0o755` octal (use `chmod 0755 filename`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by "others".

Make sure that any files your script needs to read or write are readable or writable, respectively, by "others" — their mode should be `0o644` for readable and `0o666` for writable. This is because, for security reasons, the HTTP server executes your script as user "nobody", without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (`PATH`) or the Python module search path (`PYTHONPATH`) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).

22.2.7 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

22.2.8 Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (`cgi.py`) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard `cgi-bin` directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as "addr" with

value “At Home” and “name” with value “Joe Blow”), the `cgi.py` script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the *cgi* module’s *test()* function from your script: replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the `cgi.py` file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can’t be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server’s log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute *some* code, you can easily send tracebacks to the Web browser using the *cgitb* module. If you haven’t done so already, just add the lines:

```
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the *cgitb* module, you can use an even more robust approach (which only uses built-in modules):

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

22.2.9 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client’s display while the script is running.
- Check the installation instructions above.
- Check the HTTP server’s log files. (`tail -f logfile` in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names — `PATH` is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server’s `suexec` feature.
- Don’t try to give a CGI script a set-uid mode. This doesn’t work on most systems, and is a security liability as well.

22.3 cgitb — CGI 스크립트를 위한 트레이스백 관리자

소스 코드: Lib/cgitb.py

---

cgitb 모듈은 파이썬 스크립트를 위한 특별한 예외 처리기를 제공합니다. (이 이름은 약간 오해의 소지가 있습니다. 원래는 CGI 스크립트를 위해 HTML로 광범위한 트레이스백 정보를 표시하기 위해 고안됐습니다. 나중에 이 정보를 일반 텍스트로도 표시하도록 일반화됐습니다.) 이 모듈이 활성화된 후, 잡히지 않는 예외가 발생하면, 자세한 형식의 보고서가 표시됩니다. 보고서에는 문제를 디버그하는 데 도움이 되도록, 현재 실행 중인 함수의 인자와 지역 변수의 값뿐만 아니라 각 수준의 소스 코드 발췌를 보여주는 트레이스백이 포함되어 있습니다. 선택적으로, 이 정보를 브라우저로 보내지 않고 파일에 저장할 수 있습니다.

이 기능을 활성화하려면, 단순히 CGI 스크립트 상단에 이것을 추가하면 됩니다:

```
import cgitb
cgitb.enable()
```

enable() 함수에 제공되는 옵션은 브라우저에 보고서를 표시할지와 나중에 분석 할 수 있도록 보고서를 파일에 기록할지를 제어합니다.

cgitb.**enable**(*display=1*, *logdir=None*, *context=5*, *format="html"*)

이 함수는 *cgitb* 모듈이 *sys.excepthook* 의 값을 설정하여 인터프리터의 기본 예외 처리를 인수하도록 합니다.

선택적 인자 *display* 는 기본적으로 `1` 로 설정되어 있으며 브라우저에 트레이스백을 보내지 않도록 `0` 으로 설정할 수 있습니다. *logdir* 인자가 있으면 트레이스백 보고서가 파일에 기록됩니다. *logdir* 값은 이 파일들이 위치 할 디렉터리여야 합니다. 선택적 인자 *context* 는 트레이스백에서 현재 소스 코드 행 주위에 표시할 문맥 행 수입니다. 기본값은 `5` 입니다. 선택적 인자 *format* 이 `"html"` 이면 출력은 HTML로 포맷됩니다. 그 외의 다른 값은 일반 텍스트 출력을 강제합니다. 기본값은 `"html"` 입니다.

cgitb.**text**(*info*, *context=5*)

이 함수는 *info* (*sys.exc_info()* 의 결과를 담은 3-튜플) 가 기술하는 예외를 처리하는데, 그 트레이스백을 텍스트로 포맷한 다음 결과를 문자열로 반환합니다. 선택적 인자 *context* 는 트레이스백에서 현재 소스 코드 행 주위에 표시할 문맥 행 수입니다. 기본값은 `5` 입니다.

cgitb.**html**(*info*, *context=5*)

이 함수는 *info* (*sys.exc_info()* 의 결과를 담은 3-튜플) 가 기술하는 예외를 처리하는데, 그 트레이스백을 HTML로 포맷한 다음 결과를 문자열로 반환합니다. 선택적 인자 *context* 는 트레이스백에서 현재 소스 코드 행 주위에 표시할 문맥 행 수입니다. 기본값은 `5` 입니다.

cgitb.**handler**(*info=None*)

이 함수는 기본 설정을 사용하여 예외를 처리합니다 (즉, 브라우저에 보고서를 표시하지만, 파일에 기록하지는 않습니다). 이것은 여러분이 예외를 잡았지만 *cgitb*를 사용해서 보고하고 싶을 때 사용할 수 있습니다. 선택적인 *info* 인자는 *sys.exc_info()* 에 의해 반환된 튜플과 똑같이, 예외 형, 예외 값, 트레이스백 객체를 포함하는 3-튜플이어야 합니다. *info* 인자가 제공되지 않으면, 현재 예외를 *sys.exc_info()*에서 얻습니다.

22.4 wsgiref — WSGI 유틸리티와 참조 구현

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 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.

22.4.1 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 참조하십시오.

wsgiref.util.**guess_scheme**(*environ*)
: *environ* 딕셔너리에서 HTTPS 환경 변수를 검사하여 wsgi.url_scheme이 "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 3333**의 "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 뼌한 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!

사용 예:

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
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*)
Return True if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616**.

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)
파일류 객체를 이터레이터로 변환하는 래퍼. 결과 객체는 파이썬 2.1과 Jython과의 호환성을 위해, __getitem__() 과 __iter__() 이터레이션 스타일을 모두 지원합니다. 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

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
for chunk in wrapper:
    print(chunk)
```

22.4.2 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 None을 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. None이면 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.

22.4.3 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_INFO를 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...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    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.

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 WSGI 환경을 포함하는 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.stderr를 반환합니다.

handle()
HTTP 요청을 처리합니다. 기본 구현은 *wsgiref.handlers* 클래스를 사용하여 실제 WSGI 응용 프로그램 인터페이스를 구현하는 처리기 인스턴스를 만듭니다.

22.4.4 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.errors가 아닙니다)에 기록됩니다.

사용 예:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

22.4.5 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.

버전 3.2에 추가.

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 서브클래싱하고 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 입력 스트림 객체를 반환합니다.

get_stderr()

현재 처리 중인 요청의 `wsgi.errors`로 사용하기에 적합한 출력 스트림 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. `None` 이면, 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.exc_info()`를 사용하여 현재 에러 정보에 액세스할 수 있으며, 호출할 때 해당 정보를 *start_response*로 전달해야 합니다 (**PEP 3333**의 "에러 처리" 절에서 설명하듯이).

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/plain으로 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
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.environ에서 PEP 3333 "유니코드에 들어있는 바이트열" 문자열로 변환하여, 새 딕셔너리를 반환합니다. 이 함수는 os.environ을 직접 사용하는 대신 *CGIHandler*와 *IISCGIHandler*에서 사용됩니다. os.environ은 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.environ에서 직접 값을 복사하는 대신 이 루틴을 사용하는 것이 좋습니다.

버전 3.2에 추가.

22.4.6 예제

이것은 "Hello World" WSGI 응용 프로그램입니다:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

22.5 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*

22.6 urllib.request — Extensible library for opening URLs

Source code: Lib/urllib/request.py

The *urllib.request* module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

더 보기:

The Requests package is recommended for a higher-level HTTP client interface.

The *urllib.request* module defines the following functions:

urllib.request.**urlopen**(*url*, *data=None*[, *timeout*], *, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*)

Open the URL *url*, which can be either a string or a *Request* object.

data must be an object specifying additional data to be sent to the server, or None if no such data is needed. See *Request* for details.

urllib.request module uses HTTP/1.1 and includes Connection:close header in its HTTP requests.

The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options. See *HTTPSConnection* for more details.

The optional *cafile* and *capath* parameters specify a set of trusted CA certificates for HTTPS requests. *cafile* should point to a single file containing a bundle of CA certificates, whereas *capath* should point to a directory of hashed certificate files. More information can be found in *ssl.SSLContext.load_verify_locations()*.

The *cadefault* parameter is ignored.

This function always returns an object which can work as a *context manager* and has methods such as

- `geturl()` — return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- `info()` — return the meta-information of the page, such as headers, in the form of an *email.message_from_string()* instance (see Quick Reference to HTTP Headers)
- `getcode()` – return the HTTP status code of the response.

For HTTP and HTTPS URLs, this function returns a *http.client.HTTPResponse* object slightly modified. In addition to the three new methods above, the msg attribute contains the same information as the *reason* attribute — the reason phrase returned by server — instead of the response headers as it is specified in the documentation for *HTTPResponse*.

For FTP, file, and data URLs and requests explicitly handled by legacy *URLopener* and *FancyURLopener* classes, this function returns a `urllib.response.addinfourl` object.

Raises *URLError* on protocol errors.

Note that `None` may be returned if no handler handles the request (though the default installed global *OpenerDirector* uses *UnknownHandler* to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), *ProxyHandler* is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; *urllib.request.urlopen()* corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using *ProxyHandler* objects.

버전 3.2에서 변경: *cafile* and *capath* were added.

버전 3.2에서 변경: HTTPS virtual hosts are now supported if possible (that is, if *ssl.HAS_SNI* is true).

버전 3.2에 추가: *data* can be an iterable object.

버전 3.3에서 변경: *cadefault* was added.

버전 3.4.3에서 변경: *context* was added.

버전 3.6부터 폐지: *cafile*, *capath* and *cadefault* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

`urllib.request.`**`install_opener`**(*opener*)

Install an *OpenerDirector* instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call *OpenerDirector.open()* instead of *urlopen()*. The code does not check for a real *OpenerDirector*, and any class with the appropriate interface will work.

`urllib.request.`**`build_opener`**(*[handler, ...]*)

Return an *OpenerDirector* instance, which chains the handlers in the order given. *handler*s can be either instances of *BaseHandler*, or subclasses of *BaseHandler* (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handler*s, unless the *handler*s contain them, instances of them or subclasses of them: *ProxyHandler*

(if proxy settings are detected), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor*.

If the Python installation has SSL support (i.e., if the *ssl* module can be imported), *HTTPSHandler* will also be added.

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

`urllib.request.`**`pathname2url`**(*path*)
Convert the pathname *path* from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the *quote()* function.

`urllib.request.`**`url2pathname`**(*path*)
Convert the path component *path* from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses *unquote()* to decode *path*.

`urllib.request.`**`getproxies`**()
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

참고: If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

The following classes are provided:

class `urllib.request.`**`Request`**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)
This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The *urllib.parse.urlencode()* function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

headers should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while *urllib*'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6).

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not None, `Content-Type: application/x-www-form-urlencoded` will

be added as a default.

The next two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by **RFC 2965**. It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by **RFC 2965**. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

method should be a string that indicates the HTTP request method that will be used (e.g. `'HEAD'`). If provided, its value is stored in the *method* attribute and is used by *get_method()*. The default is `'GET'` if *data* is `None` or `'POST'` otherwise. Subclasses may indicate a different default method by setting the *method* attribute in the class itself.

참고: The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

버전 3.3에서 변경: *Request.method* argument is added to the Request class.

버전 3.4에서 변경: Default *Request.method* may be indicated at the class level.

버전 3.6에서 변경: Do not raise an error if the `Content-Length` has not been provided and *data* is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class urllib.request.**OpenerDirector**
The *OpenerDirector* class opens URLs via *BaseHandler*s chained together. It manages the chaining of handlers, and recovery from errors.

class urllib.request.**BaseHandler**
This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class urllib.request.**HTTPDefaultErrorHandler**
A class which defines a default handler for HTTP error responses; all responses are turned into *HTTPError* exceptions.

class urllib.request.**HTTPRedirectHandler**
A class to handle redirections.

class urllib.request.**HTTPCookieProcessor**(*cookiejar=None*)
A class to handle HTTP Cookies.

class urllib.request.**ProxyHandler**(*proxies=None*)
Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The `no_proxy` environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

참고: HTTP_PROXY will be ignored if a variable REQUEST_METHOD is set; see the documentation on *getproxies()*.

class urllib.request.**HTTPPasswordMgr**
Keep a database of (realm, uri) -> (user, password) mappings.

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**
Keep a database of (realm, uri) -> (user, password) mappings. A realm of None is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.**HTTPPasswordMgrWithPriorAuth**
A variant of *HTTPPasswordMgrWithDefaultRealm* that also has a database of uri -> is_authenticated mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a 401 response first.

버전 3.5에 추가.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. If *passwd_mgr* also provides is_authenticated and update_authenticated methods (see *HTTPPasswordMgrWithPriorAuth Objects*), then the handler will use the is_authenticated result for a given URI to determine whether or not to send authentication credentials with the request. If is_authenticated returns True for the URI, credentials are sent. If is_authenticated is False, credentials are not sent, and then if a 401 response is received the request is re-sent with the authentication credentials. If authentication succeeds, update_authenticated is called to set is_authenticated True for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

버전 3.5에 추가: Added is_authenticated support.

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. HTTPBasicAuthHandler will raise a *ValueError* when presented with a wrong Authentication scheme.

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a *ValueError* when presented with an authentication scheme other than Digest or Basic.

버전 3.3에서 변경: Raise *ValueError* on unsupported Authentication Scheme.

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**HTTPHandler**
A class to handle opening of HTTP URLs.

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)
A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in *http.client.HTTPSConnection*.

버전 3.2에서 변경: *context* and *check_hostname* were added.

class urllib.request.**FileHandler**
Open local files.

class urllib.request.**DataHandler**
Open data URLs.

버전 3.4에 추가.

class urllib.request.**FTPHandler**
Open FTP URLs.

class urllib.request.**CacheFTPHandler**
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.**UnknownHandler**
A catch-all class to handle unknown URLs.

class urllib.request.**HTTPErrorProcessor**
Process HTTP error responses.

22.6.1 Request Objects

The following methods describe *Request*'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.**full_url**
The original URL passed to the constructor.

버전 3.4에서 변경.

Request.full_url is a property with setter, getter and a deleter. Getting *full_url* returns the original request URL with the fragment, if it was present.

Request.**type**
The URI scheme.

Request.**host**
The URI authority, typically a host, but may also contain a port separated by a colon.

Request.**origin_req_host**
The original host for the request, without port.

Request.**selector**
The URI path. If the *Request* uses a proxy, then selector will be the full URL that is passed to the proxy.

Request.**data**
The entity body for the request, or None if not specified.

버전 3.4에서 변경: Changing value of *Request.data* now deletes "Content-Length" header if it was previously set or calculated.

Request.**unverifiable**
boolean, indicates whether the request is unverifiable as defined by **RFC 2965**.

Request.**method**
The HTTP request method to use. By default its value is *None*, which means that *get_method()* will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in *get_method()*) either by providing a default value by setting it at the class level in a *Request* subclass, or by passing a value in to the *Request* constructor via the *method* argument.

버전 3.3에 추가.

버전 3.4에서 변경: A default value can now be set in subclasses; previously it could only be set via the constructor argument.

Request.**get_method**()
Return a string indicating the HTTP request method. If *Request.method* is not None, return its value, otherwise return 'GET' if *Request.data* is None, or 'POST' if it's not. This is only meaningful for HTTP requests.

버전 3.3에서 변경: get_method now looks at the value of *Request.method*.

Request.**add_header**(*key*, *val*)
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

Request.**add_unredirected_header**(*key*, *header*)
Add a header that will not be added to a redirected request.

Request.**has_header**(*header*)
Return whether the instance has the named header (checks both regular and unredirected).

Request.**remove_header**(*header*)
Remove named header from the request instance (both from regular and unredirected headers).

버전 3.4에 추가.

Request.**get_full_url**()
Return the URL given in the constructor.

버전 3.4에서 변경.

Returns *Request.full_url*

Request.**set_proxy**(*host*, *type*)
Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

Request.**get_header**(*header_name*, *default=None*)
Return the value of the given header. If the header is not present, return the default value.

Request.**header_items**()
Return a list of tuples (header_name, header_value) of the Request headers.

버전 3.4에서 변경: The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.

22.6.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

OpenerDirector.**add_handler**(*handler*)

handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case). Note that, in the following, *protocol* should be replaced with the actual protocol to handle, for example `http_response()` would be the HTTP protocol response handler. Also *type* should be replaced with the actual HTTP code, for example `http_error_404()` would handle HTTP 404 errors.

- `<protocol>_open()` — signal that the handler knows how to open *protocol* URLs.

  See *BaseHandler.<protocol>_open()* for more information.

- `http_error_<type>()` — signal that the handler knows how to handle HTTP errors with HTTP error code *type*.

  See *BaseHandler.http_error_<nnn>()* for more information.

- `<protocol>_error()` — signal that the handler knows how to handle errors from (non-`http`) *protocol*.

- `<protocol>_request()` — signal that the handler knows how to pre-process *protocol* requests.

  See *BaseHandler.<protocol>_request()* for more information.

- `<protocol>_response()` — signal that the handler knows how to post-process *protocol* responses.

  See *BaseHandler.<protocol>_response()* for more information.

OpenerDirector.**open**(*url*, *data=None*[, *timeout*])

Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of *urlopen()* (which simply calls the *open()* method on the currently installed global *OpenerDirector*). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections).

OpenerDirector.**error**(*proto*, **args*)

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the `http_error_<type>()` methods of the handler classes.

Return values and exceptions raised are the same as those of *urlopen()*.

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `<protocol>_request()` has that method called to pre-process the request.

2. Handlers with a method named like `<protocol>_open()` are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named `default_open()`. If all such methods return *None*, the algorithm is repeated for methods named like `<protocol>_open()`. If all such methods return *None*, the algorithm is repeated for methods named `unknown_open()`.

   Note that the implementation of these methods may involve calls of the parent *OpenerDirector* instance's *open()* and *error()* methods.

3. Every handler with a method named like `<protocol>_response()` has that method called to post-process the response.

22.6.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

`BaseHandler.`**`add_parent`**`(`*director*`)`
Add a director as parent.

`BaseHandler.`**`close`**`()`
Remove any parents.

The following attribute and methods should only be used by classes derived from *BaseHandler*.

참고: The convention has been adopted that subclasses defining `<protocol>_request()` or `<protocol>_response()` methods are named `*Processor`; all others are named `*Handler`.

`BaseHandler.`**`parent`**
A valid *OpenerDirector*, which can be used to open using a different protocol, or handle errors.

`BaseHandler.`**`default_open`**`(`*req*`)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* of *OpenerDirector*, or `None`. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to `URLError`).

This method will be called before any protocol-specific open method.

`BaseHandler.<protocol>_open(req)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. Return values should be the same as for `default_open()`.

`BaseHandler.`**`unknown_open`**`(`*req*`)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the *parent OpenerDirector*. Return values should be the same as for *default_open()*.

`BaseHandler.`**`http_error_default`**`(`*req*, *fp*, *code*, *msg*, *hdrs*`)`
This method is *not* defined in *BaseHandler*, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the *OpenerDirector* getting the error, and should not normally be called in other circumstances.

req will be a *Request* object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of *urlopen()*.

`BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)`
nnn should be a three-digit HTTP error code. This method is also not defined in *BaseHandler*, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for `http_error_default()`.

BaseHandler.<protocol>_request(req)
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. The return value should be a *Request* object.

BaseHandler.<protocol>_response(req, response)
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. *response* will be an object implementing the same interface as the return value of *urlopen()*. The return value should implement the same interface as the return value of *urlopen()*.

22.6.4 HTTPRedirectHandler Objects

참고: Some HTTP redirections require action from this module's client code. If this is the case, *HTTPError* is raised. See **RFC 2616** for details of the precise meanings of the various redirection codes.

An `HTTPError` exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

`HTTPRedirectHandler.`**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)
Return a *Request* or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new *Request* to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise, raise *HTTPError* if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

참고: The default implementation of this method does not strictly follow **RFC 2616**, which says that 301 and 302 responses to `POST` requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a `GET`, and the default implementation reproduces this behavior.

`HTTPRedirectHandler.`**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)
Redirect to the `Location:` or `URI:` URL. This method is called by the parent *OpenerDirector* when getting an HTTP 'moved permanently' response.

`HTTPRedirectHandler.`**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'found' response.

`HTTPRedirectHandler.`**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'see other' response.

`HTTPRedirectHandler.`**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'temporary redirect' response.

22.6.5 HTTPCookieProcessor Objects

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.**cookiejar**
The *http.cookiejar.CookieJar* in which cookies are stored.

22.6.6 ProxyHandler Objects

ProxyHandler.<protocol>_open(request)
The *ProxyHandler* will have a method `<protocol>_open()` for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

22.6.7 HTTPPasswordMgr Objects

These methods are available on *HTTPPasswordMgr* and *HTTPPasswordMgrWithDefaultRealm* objects.

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)
uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For *HTTPPasswordMgrWithDefaultRealm* objects, the realm `None` will be searched if the given *realm* has no matching user/password.

22.6.8 HTTPPasswordMgrWithPriorAuth Objects

This password manager extends *HTTPPasswordMgrWithDefaultRealm* to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)
realm, *uri*, *user*, *passwd* are as for *HTTPPasswordMgr.add_password()*. *is_authenticated* sets the initial value of the `is_authenticated` flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
Same as for *HTTPPasswordMgrWithDefaultRealm* objects

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)
Update the `is_authenticated` flag for the given *uri* or list of URIs.

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)
Returns the current state of the `is_authenticated` flag for the given URI.

22.6.9 AbstractBasicAuthHandler Objects

`AbstractBasicAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)
Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

22.6.10 HTTPBasicAuthHandler Objects

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.11 ProxyBasicAuthHandler Objects

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.12 AbstractDigestAuthHandler Objects

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)
authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

22.6.13 HTTPDigestAuthHandler Objects

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.14 ProxyDigestAuthHandler Objects

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.15 HTTPHandler Objects

`HTTPHandler.`**`http_open`**(*req*)
Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

22.6.16 HTTPSHandler Objects

`HTTPSHandler.`**`https_open`**`(req)`
Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

22.6.17 FileHandler Objects

`FileHandler.`**`file_open`**`(req)`
Open the file locally, if there is no host name, or the host name is `'localhost'`.

버전 3.2에서 변경: This method is applicable only for local hostnames. When a remote hostname is given, an *URLError* is raised.

22.6.18 DataHandler Objects

`DataHandler.`**`data_open`**`(req)`
Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in **RFC 2397**. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise an *ValueError* in that case.

22.6.19 FTPHandler Objects

`FTPHandler.`**`ftp_open`**`(req)`
Open the FTP file indicated by *req*. The login is always done with empty username and password.

22.6.20 CacheFTPHandler Objects

CacheFTPHandler objects are *FTPHandler* objects with the following additional methods:

`CacheFTPHandler.`**`setTimeout`**`(t)`
Set timeout of connections to *t* seconds.

`CacheFTPHandler.`**`setMaxConns`**`(m)`
Set maximum number of cached connections to *m*.

22.6.21 UnknownHandler Objects

`UnknownHandler.`**`unknown_open`**`()`
Raise a *URLError* exception.

22.6.22 HTTPErrorProcessor Objects

`HTTPErrorProcessor.`**`http_response`**(*request*, *response*)
Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the `http_error_<type>()` handler methods, via *`OpenerDirector.error()`*. Eventually, *`HTTPDefaultErrorHandler`* will raise an *`HTTPError`* if no other handler handles the error.

`HTTPErrorProcessor.`**`https_response`**(*request*, *response*)
Process HTTPS error responses.

The behavior is same as *`http_response()`*.

22.6.23 Examples

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 300 bytes of it.

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following W3C document, https://www.w3.org/International/O-charset, lists the various ways in which an (X)HTML or an XML document could have specified its encoding information.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object.

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

It is also possible to achieve the same result without using the *context manager* approach.

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> 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"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a `PUT` request using *Request*:

```
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)
```

Use of Basic HTTP Authentication:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
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)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() provides many handlers by default, including a *ProxyHandler*. By default, *ProxyHandler* uses the environment variables named <scheme>_proxy, where <scheme> is the URL scheme involved. For example, the `http_proxy` environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default *ProxyHandler* with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with *ProxyBasicAuthHandler*.

```
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)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers:

Use the *headers* argument to the *Request* constructor, or:

```
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: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector automatically adds a *User-Agent* header to every *Request*. To change this:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).

Here is an example session that uses the `GET` method to retrieve a URL containing parameters:

```
>>> 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'))
...
```

The following example uses the `POST` method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```
>>> 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'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

22.6.24 Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.`**`urlretrieve`**(*url*, *filename=None*, *reporthook=None*, *data=None*)

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by *urlopen()* returned (for a remote object). Exceptions are the same as for *urlopen()*.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

urlretrieve() will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.`**`urlcleanup`**()

Cleans up temporary files that may have been left behind by previous calls to *urlretrieve()*.

`class` `urllib.request.`**`URLopener`**(*proxies=None*, ***x509*)

버전 3.3부터 폐지.

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use *FancyURLopener*.

By default, the *URLopener* class sends a *User-Agent* header of `urllib/VVV`, where *VVV* is the *urllib* version number. Applications can define their own *User-Agent* header by subclassing *URLopener* or *FancyURLopener* and setting the class attribute *version* to an appropriate string value in the subclass definition.

The optional *proxies* parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of *urlopen()*, above.

Additional keyword parameters, collected in *x509*, may be used for authentication of the client when using the `https:` scheme. The keywords *key_file* and *cert_file* are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an *OSError* exception if the server returns an error code.

open(*fullurl*, *data=None*)
Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, *open_unknown()* is called. The *data* argument has the same meaning as the *data* argument of *urlopen()*.

This method always quotes *fullurl* using *quote()*.

open_unknown(*fullurl*, *data=None*)
Overridable interface to open unknown URL types.

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)
Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either an *email.message.Message* object containing the response headers (for remote URLs) or `None` (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of *tempfile.mktemp()* with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

version
Variable that specifies the user agent of the opener object. To get *urllib* to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.request.**FancyURLopener**(...)
버전 3.3부터 폐지.

FancyURLopener subclasses *URLopener* providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method `http_error_default()` is called which you can override in subclasses to handle the error appropriately.

참고: According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and *urllib* reproduces this behaviour.

The parameters to the constructor are the same as those for *URLopener*.

참 고 : When performing basic authentication, a *FancyURLopener* instance calls its *prompt_user_passwd()* method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The *FancyURLopener* class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(*host*, *realm*)
Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, `(user, password)`, which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

22.6.25 urllib.request Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  버전 3.4에서 변경: Added support for data URLs.

- The caching feature of *urlretrieve()* has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The *urlopen()* and *urlretrieve()* functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by *urlopen()* or *urlretrieve()* is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *Content-Type* header. If the returned data is HTML, you can use the module *html.parser* to parse it.
- 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, subclassing *FancyURLopener*, or changing *_urlopener* to meet your needs.

22.7 urllib.response — Response classes used by urllib

The *urllib.response* module defines functions and classes which define a minimal file like interface, including `read()` and `readline()`. The typical response object is an addinfourl instance, which defines an `info()` method and that returns headers and a `geturl()` method that returns the url. Functions defined by this module are used internally by the *urllib.request* module.

22.8 urllib.parse — Parse URLs into components

Source code: Lib/urllib/parse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

The *urllib.parse* module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

22.8.1 URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

urllib.parse.**urlparse**(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-item *named tuple*. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in **RFC 1808**, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> 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='')
```

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring*, except that the default value `''` is always allowed, and is automatically converted to `b''` if appropriate.

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is a *named tuple*, which means that its items can be accessed by index or as named attributes, which are:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`params`	3	Parameters for last path element	empty string
`query`	4	Query component	empty string
`fragment`	5	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

As is the case with all named tuples, the subclass has a few additional methods and attributes that are particularly useful. One such method is `_replace()`. The `_replace()` method will return a new ParseResult object replacing specified fields with new values.

```
>>> 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: Added IPv6 URL parsing capabilities.

버전 3.3에서 변경: The fragment is now parsed for all URL schemes (unless *allow_fragment* is false), in accordance with **RFC 3986**. Previously, a whitelist of schemes that support fragments existed.

버전 3.6에서 변경: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

버전 3.7.3에서 변경: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

urllib.parse.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the *urllib.parse.urlencode()* function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

버전 3.2에서 변경: Add *encoding* and *errors* parameters.

버전 3.7.2에서 변경: Added *max_num_fields* parameter.

버전 3.7.10에서 변경: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.7.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urllib.parse.**parse_qsl**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the *urllib.parse.urlencode()* function to convert such lists of pairs into query strings.

버전 3.2에서 변경: Add *encoding* and *errors* parameters.

버전 3.7.2에서 변경: Added *max_num_fields* parameter.

버전 3.7.10에서 변경: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.7.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urllib.parse.**urlunparse**(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable.

This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

urllib.parse.**urlsplit**(*urlstring*, *scheme=''*, *allow_fragments=True*)

This is similar to *urlparse()*, but does not split the params from the URL. This should generally be used instead of *urlparse()* if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-item *named tuple*:

```
(addressing scheme, network location, path, query, fragment identifier).
```

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

Attribute	Index	Value	Value if not present
scheme	0	URL scheme specifier	*scheme* parameter
netloc	1	Network location part	empty string
path	2	Hierarchical path	empty string
query	3	Query component	empty string
fragment	4	Fragment identifier	empty string
username		User name	*None*
password		Password	*None*
hostname		Host name (lower case)	*None*
port		Port number as integer, if present	*None*

Reading the port attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the netloc attribute will raise a *ValueError*.

Characters in the netloc attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of /, ?, #, @, or : will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

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: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

버전 3.7.3에서 변경: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

버전 3.7.11에서 변경: ASCII newline and tab characters are stripped from the URL.

버전 3.7.17에서 변경: Leading WHATWG C0 control and space characters are stripped from the URL.

urllib.parse.**urlunsplit**(*parts*)

Combine the elements of a tuple as returned by *urlsplit()* into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

urllib.parse.**urljoin**(*base*, *url*, *allow_fragments=True*)

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for *urlparse()*.

참고: If *url* is an absolute URL (that is, starting with `//` or `scheme://`), the *url*'s host name and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with *urlsplit()* and *urlunsplit()*, removing possible *scheme* and *netloc* parts.

버전 3.5에서 변경: Behaviour updated to match the semantics defined in **RFC 3986**.

`urllib.parse.`**`urldefrag`**(*url*)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

Attribute	Index	Value	Value if not present
`url`	0	URL with no fragment	empty string
`fragment`	1	Fragment identifier	empty string

See section *Structured Parse Results* for more information on the result object.

버전 3.2에서 변경: Result is a structured object rather than a simple 2-tuple.

22.8.2 URL parsing security

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.

22.8.3 Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on *bytes* and *bytearray* objects in addition to *str* objects.

If *str* data is passed in, the result will also contain only *str* data. If *bytes* or *bytearray* data is passed in, the result will contain only *bytes* data.

Attempting to mix *str* data with *bytes* or *bytearray* in a single function call will result in a *TypeError* being raised, while attempting to pass in non-ASCII byte values will trigger *UnicodeDecodeError*.

To support easier conversion of result objects between *str* and *bytes*, all return values from URL parsing functions provide either an `encode()` method (when the result contains *str* data) or a `decode()` method (when the result contains *bytes* data). The signatures of these methods match those of the corresponding *str* and *bytes* methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either *bytes* data (for `encode()` methods) or *str* data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

버전 3.2에서 변경: URL parsing functions now accept ASCII encoded byte sequences

22.8.4 Structured Parse Results

The result objects from the *urlparse()*, *urlsplit()* and *urldefrag()* functions are subclasses of the *tuple* type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

`urllib.parse.SplitResult.`**`geturl`**`()`

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For *urldefrag()* results, only empty fragment identifiers will be removed. For *urlsplit()* and *urlparse()* results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```
>>> 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/'
```

The following classes provide the implementations of the structured parse results when operating on *str* objects:

`class` `urllib.parse.`**`DefragResult`**`(`*url*, *fragment*`)`

Concrete class for *urldefrag()* results containing *str* data. The `encode()` method returns a *DefragResultBytes* instance.

버전 3.2에 추가.

class urllib.parse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results containing *str* data. The encode() method returns a *ParseResultBytes* instance.

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results containing *str* data. The encode() method returns a *SplitResultBytes* instance.

The following classes provide the implementations of the parse results when operating on *bytes* or *bytearray* objects:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)
Concrete class for *urldefrag()* results containing *bytes* data. The decode() method returns a *DefragResult* instance.

버전 3.2에 추가.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results containing *bytes* data. The decode() method returns a *ParseResult* instance.

버전 3.2에 추가.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results containing *bytes* data. The decode() method returns a *SplitResult* instance.

버전 3.2에 추가.

22.8.5 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)
Replace special characters in *string* using the %xx escape. Letters, digits, and the characters '_.-~' are never quoted. By default, this function is intended for quoting the path section of URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted — its default value is '/'.

string may be either a *str* or a *bytes*.

버전 3.7에서 변경: Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. "~" is now included in the set of unreserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the *str.encode()* method. *encoding* defaults to 'utf-8'. *errors* defaults to 'strict', meaning unsupported characters raise a *UnicodeEncodeError*. *encoding* and *errors* must not be supplied if *string* is a *bytes*, or a *TypeError* is raised.

Note that quote(string, safe, encoding, errors) is equivalent to quote_from_bytes(string.encode(encoding, errors), safe).

Example: quote('/El Niño/') yields '/El%20Ni%C3%B1o/'.

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)
Like *quote()*, but also replace spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to '/'.

Example: `quote_plus('/El Niño/')` yields `'%2FEl+Ni%C3%B1o%2F'`.

`urllib.parse.`**`quote_from_bytes`**(*bytes*, *safe='/'*)
Like *quote()*, but accepts a *bytes* object rather than a *str*, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

`urllib.parse.`**`unquote`**(*string*, *encoding='utf-8'*, *errors='replace'*)
Replace `%xx` escapes by their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

string must be a *str*.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.`**`unquote_plus`**(*string*, *encoding='utf-8'*, *errors='replace'*)
Like *unquote()*, but also replace plus signs by spaces, as required for unquoting HTML form values.

string must be a *str*.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.`**`unquote_to_bytes`**(*string*)
Replace `%xx` escapes by their single-octet equivalent, and return a *bytes* object.

string may be either a *str* or a *bytes*.

If it is a *str*, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

`urllib.parse.`**`urlencode`**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)
Convert a mapping object or a sequence of two-element tuples, which may contain *str* or *bytes* objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the *urlopen()* function, then it should be encoded to bytes, otherwise it would result in a *TypeError*.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, *quote_plus()* is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is *quote()*, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* is evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a *str*).

To reverse this encoding process, *parse_qs()* and *parse_qsl()* are provided in this module to parse query strings into Python data structures.

Refer to *urllib examples* to find out how urlencode method can be used for generating query string for a URL or data for POST.

버전 3.2에서 변경: Query parameter supports bytes and string objects.

버전 3.5에 추가: *quote_via* parameter.

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 This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's. This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL) This specifies the formal syntax and semantics of absolute URLs.

22.9 `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: *URLError*는 *IOError*가 아닌, *OSError*의 서브 클래스가 되었습니다.

exception urllib.error.**HTTPError**
*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.

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.

headers
*HTTPError*를 발생시킨 HTTP 요청의 응답 헤더입니다.

버전 3.4에 추가.

exception urllib.error.**ContentTooShortError**(*msg*, *content*)
이 예외는 다운로드받은 데이터양이 *Content-Length* 헤더 값을 통해 예상한 양보다 적은 것을 *urlretrieve()*함수가 감지했을 때 발생합니다. content어트리뷰트는 다운로드받은 (그리고 아마도 잘린) 데이터를 저장합니다.

22.10 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.

버전 3.6에 추가.

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.

버전 3.6에 추가.

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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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
```

22.11 `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

*http*는 여러 HTTP 상태 코드와 관련 메시지를 *http.HTTPStatus* 열거형을 통해 정의하는 모듈이기도 합니다:

class http.**HTTPStatus**

버전 3.5에 추가.

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: 200>
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

22.11.1 HTTP 상태 코드

*http.HTTPStatus*에서 지원하는 IANA 등록 상태 코드는 다음과 같습니다:

코드	열거 이름	세부 사항
100	CONTINUE	HTTP/1.1 **RFC 7231**, 섹션 6.2.1
101	SWITCHING_PROTOCOLS	HTTP/1.1 **RFC 7231**, 섹션 6.2.2
102	PROCESSING	WebDAV **RFC 2518**, 섹션 10.1
200	OK	HTTP/1.1 **RFC 7231**, 섹션 6.3.1
201	CREATED	HTTP/1.1 **RFC 7231**, 섹션 6.3.2
202	ACCEPTED	HTTP/1.1 **RFC 7231**, 섹션 6.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 **RFC 7231**, 섹션 6.3.4
204	NO_CONTENT	HTTP/1.1 **RFC 7231**, 섹션 6.3.5
205	RESET_CONTENT	HTTP/1.1 **RFC 7231**, 섹션 6.3.6
206	PARTIAL_CONTENT	HTTP/1.1 **RFC 7233**, 섹션 4.1
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/1.1 **RFC 7231**, 섹션 6.4.1
301	MOVED_PERMANENTLY	HTTP/1.1 **RFC 7231**, 섹션 6.4.2
302	FOUND	HTTP/1.1 **RFC 7231**, 섹션 6.4.3
303	SEE_OTHER	HTTP/1.1 **RFC 7231**, 섹션 6.4.4
304	NOT_MODIFIED	HTTP/1.1 **RFC 7232**, 섹션 4.1
305	USE_PROXY	HTTP/1.1 **RFC 7231**, 섹션 6.4.5
307	TEMPORARY_REDIRECT	HTTP/1.1 **RFC 7231**, 섹션 6.4.7
308	PERMANENT_REDIRECT	영구 리디렉션 **RFC 7238**, 섹션 3 (실험적)
400	BAD_REQUEST	HTTP/1.1 **RFC 7231**, 섹션 6.5.1
401	UNAUTHORIZED	HTTP/1.1 인증 **RFC 7235**, 섹션 3.1
402	PAYMENT_REQUIRED	HTTP/1.1 **RFC 7231**, 섹션 6.5.2
403	FORBIDDEN	HTTP/1.1 **RFC 7231**, 섹션 6.5.3
404	NOT_FOUND	HTTP/1.1 **RFC 7231**, 섹션 6.5.4
405	METHOD_NOT_ALLOWED	HTTP/1.1 **RFC 7231**, 섹션 6.5.5
406	NOT_ACCEPTABLE	HTTP/1.1 **RFC 7231**, 섹션 6.5.6
407	PROXY_AUTHENTICATION_REQUIRED	HTTP/1.1 인증 **RFC 7235**, 섹션 3.2
408	REQUEST_TIMEOUT	HTTP/1.1 **RFC 7231**, 섹션 6.5.7
409	CONFLICT	HTTP/1.1 **RFC 7231**, 섹션 6.5.8
410	GONE	HTTP/1.1 **RFC 7231**, 섹션 6.5.9
411	LENGTH_REQUIRED	HTTP/1.1 **RFC 7231**, 섹션 6.5.10
412	PRECONDITION_FAILED	HTTP/1.1 **RFC 7232**, 섹션 4.2
413	REQUEST_ENTITY_TOO_LARGE	HTTP/1.1 **RFC 7231**, 섹션 6.5.11
414	REQUEST_URI_TOO_LONG	HTTP/1.1 **RFC 7231**, 섹션 6.5.12
415	UNSUPPORTED_MEDIA_TYPE	HTTP/1.1 **RFC 7231**, 섹션 6.5.13
416	REQUESTED_RANGE_NOT_SATISFIABLE	HTTP/1.1 범위 요청 **RFC 7233**, 섹션 4.4
417	EXPECTATION_FAILED	HTTP/1.1 **RFC 7231**, 섹션 6.5.14
421	MISDIRECTED_REQUEST	HTTP/2 **RFC 7540**, 섹션 9.1.2
422	UNPROCESSABLE_ENTITY	WebDAV **RFC 4918**, 섹션 11.2
423	LOCKED	WebDAV **RFC 4918**, 섹션 11.3
424	FAILED_DEPENDENCY	WebDAV **RFC 4918**, 섹션 11.4
426	UPGRADE_REQUIRED	HTTP/1.1 **RFC 7231**, 섹션 6.5.15
428	PRECONDITION_REQUIRED	추가 HTTP 상태 코드 **RFC 6585**
429	TOO_MANY_REQUESTS	추가 HTTP 상태 코드 **RFC 6585**

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

코드	열거 이름	세부 사항
431	REQUEST_HEADER_FIELDS_TOO_LARGE	추가 HTTP 상태 코드 **RFC 6585**
500	INTERNAL_SERVER_ERROR	HTTP/1.1 **RFC 7231**, 섹션 6.6.1
501	NOT_IMPLEMENTED	HTTP/1.1 **RFC 7231**, 섹션 6.6.2
502	BAD_GATEWAY	HTTP/1.1 **RFC 7231**, 섹션 6.6.3
503	SERVICE_UNAVAILABLE	HTTP/1.1 **RFC 7231**, 섹션 6.6.4
504	GATEWAY_TIMEOUT	HTTP/1.1 **RFC 7231**, 섹션 6.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP/1.1 **RFC 7231**, 섹션 6.6.6
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.

22.12 `http.client` — HTTP protocol client

Source code: Lib/http/client.py

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module *urllib.request* uses it to handle URLs that use HTTP and HTTPS.

더 보기:

The Requests package is recommended for a higher-level HTTP client interface.

참고: HTTPS support is only available if Python was compiled with SSL support (through the *ssl* module).

The module provides the following classes:

class http.client.**HTTPConnection**(*host*, *port=None*[, *timeout*], *source_address=None*, *blocksize=8192*)

An *HTTPConnection* instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from. The optional *blocksize* parameter sets the buffer size in bytes for sending a file-like message body.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> 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* was added.

버전 3.4에서 변경: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are not longer supported.

버전 3.7에서 변경: *blocksize* parameter was added.

class http.client.**HTTPSConnection**(*host*, *port=None*, *key_file=None*, *cert_file=None*[, *timeout*], *source_address=None*, *, *context=None*, *check_hostname=None*, *blocksize=8192*)

A subclass of *HTTPConnection* that uses SSL for communication with secure servers. Default port is `443`. If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options.

Please read 보안 고려 사항 for more information on best practices.

버전 3.2에서 변경: *source_address*, *context* and *check_hostname* were added.

버전 3.2에서 변경: This class now supports HTTPS virtual hosts if possible (that is, if *ssl.HAS_SNI* is true).

버전 3.4에서 변경: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

버전 3.4.3에서 변경: This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior `ssl._create_unverified_context()` can be passed to the *context* parameter.

버전 3.7.4에서 변경: This class now enables TLS 1.3 *ssl.SSLContext.post_handshake_auth* for the default *context* or when *cert_file* is passed with a custom *context*.

버전 3.6부터 폐지: *key_file* and *cert_file* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

The *check_hostname* parameter is also deprecated; the *ssl.SSLContext.check_hostname* attribute of *context* should be used instead.

class http.client.**HTTPResponse**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

버전 3.4에서 변경: The *strict* parameter was removed. HTTP 0.9 style "Simple Responses" are no longer supported.

The following exceptions are raised as appropriate:

exception http.client.**HTTPException**

The base class of the other exceptions in this module. It is a subclass of *Exception*.

exception http.client.**NotConnected**

A subclass of *HTTPException*.

exception http.client.**InvalidURL**

A subclass of *HTTPException*, raised if a port is given and is either non-numeric or empty.

exception http.client.**UnknownProtocol**

A subclass of *HTTPException*.

exception http.client.**UnknownTransferEncoding**

A subclass of *HTTPException*.

exception http.client.**UnimplementedFileMode**

A subclass of *HTTPException*.

exception http.client.**IncompleteRead**

A subclass of *HTTPException*.

exception http.client.**ImproperConnectionState**
A subclass of *HTTPException*.

exception http.client.**CannotSendRequest**
A subclass of *ImproperConnectionState*.

exception http.client.**CannotSendHeader**
A subclass of *ImproperConnectionState*.

exception http.client.**ResponseNotReady**
A subclass of *ImproperConnectionState*.

exception http.client.**BadStatusLine**
A subclass of *HTTPException*. Raised if a server responds with a HTTP status code that we don't understand.

exception http.client.**LineTooLong**
A subclass of *HTTPException*. Raised if an excessively long line is received in the HTTP protocol from the server.

exception http.client.**RemoteDisconnected**
A subclass of *ConnectionResetError* and *BadStatusLine*. Raised by *HTTPConnection.getresponse()* when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

버전 3.5에 추가: Previously, *BadStatusLine*('') was raised.

The constants defined in this module are:

http.client.**HTTP_PORT**
The default port for the HTTP protocol (always 80).

http.client.**HTTPS_PORT**
The default port for the HTTPS protocol (always 443).

http.client.**responses**
This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: http.client.responses[http.client.NOT_FOUND] is 'Not Found'.

See *HTTP 상태 코드* for a list of HTTP status codes that are available in this module as constants.

22.12.1 HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)
This will send a request to the server using the HTTP request method *method* and the selector *url*.

If *body* is specified, the specified data is sent after the headers are finished. It may be a *str*, a *bytes-like object*, an open *file object*, or an iterable of *bytes*. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a *file object*, the contents of the file is sent; this file object should support at least the read() method. If the file object is an instance of *io.TextIOBase*, the data returned by the read() method will be encoded as ISO-8859-1, otherwise the data returned by read() is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

The *headers* argument should be a mapping of extra HTTP headers to send with the request.

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is None, the Content-Length header is set to 0 for methods that expect a body (PUT, POST, and PATCH). If *body* is a string or a bytes-like object that is not also a *file*, the Content-Length

header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

참고: Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a *str* or bytes-like object that is not also a file as the body representation.

버전 3.2에 추가: *body* can now be an iterable.

버전 3.6에서 변경: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

`HTTPConnection.`**`getresponse`**`()`
Should be called after a request is sent to get the response from the server. Returns an *HTTPResponse* instance.

참고: Note that you must have read the whole response before you can send a new request to the server.

버전 3.5에서 변경: If a *ConnectionError* or subclass is raised, the *HTTPConnection* object will be ready to reconnect when a new request is sent.

`HTTPConnection.`**`set_debuglevel`**`(`*level*`)`
Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new *HTTPResponse* objects that are created.

버전 3.1에 추가.

`HTTPConnection.`**`set_tunnel`**`(`*host*, *port=None*, *headers=None*`)`
Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server.

The host and port arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the *HTTPSConnection* constructor, and the address of the host that we eventually want to reach to the *set_tunnel()* method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

버전 3.2에 추가.

`HTTPConnection.`**`connect`**`()`
Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

`HTTPConnection.`**`close`**`()`
Close the connection to the server.

HTTPConnection.**blocksize**
Buffer size in bytes for sending a file-like message body.

버전 3.7에 추가.

As an alternative to using the `request()` method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.**putrequest**(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (`HTTP/1.1`). To disable automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

HTTPConnection.**putheader**(*header*, *argument*[, ...])
Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.**endheaders**(*message_body=None*, *, *encode_chunked=False*)
Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a *collections.abc.Iterable*, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

참고: Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

버전 3.6에 추가: Chunked encoding support. The *encode_chunked* parameter was added.

HTTPConnection.**send**(*data*)
Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

22.12.2 HTTPResponse Objects

An *HTTPResponse* instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

버전 3.5에서 변경: The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

HTTPResponse.**read**([*amt*])
Reads and returns the response body, or up to the next *amt* bytes.

HTTPResponse.**readinto**(*b*)
Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

버전 3.3에 추가.

HTTPResponse.**getheader**(*name*, *default=None*)
Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one

header with the name *name*, return all of the values joined by ', '. If 'default' is any iterable other than a single string, its elements are similarly returned joined by commas.

HTTPResponse.**getheaders**()
Return a list of (header, value) tuples.

HTTPResponse.**fileno**()
Return the `fileno` of the underlying socket.

HTTPResponse.**msg**
A `http.client.HTTPMessage` instance containing the response headers. `http.client.HTTPMessage` is a subclass of *email.message.Message*.

HTTPResponse.**version**
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.**status**
Status code returned by server.

HTTPResponse.**reason**
Reason phrase returned by server.

HTTPResponse.**debuglevel**
A debugging hook. If *debuglevel* is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse.**closed**
Is `True` if the stream is closed.

22.12.3 Examples

Here is an example session that uses the `GET` method:

```
>>> 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()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while True:
...     chunk = r1.read(200)  # 200 bytes
...     if not chunk:
...         break
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> 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()
```

Here is an example session that uses the HEAD method. Note that the HEAD method never returns any data.

```
>>> 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
```

Here is an example session that shows how to POST requests:

```
>>> 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()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

Client side HTTP PUT requests are very similar to POST requests. The difference lies only the server side where HTTP server will allow resources to be created via PUT request. It should be noted that custom HTTP methods are also handled in *urllib.request.Request* by setting the appropriate method attribute. Here is an example session that shows how to send a PUT request using http.client:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> 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
```

22.12.4 HTTPMessage Objects

An `http.client.HTTPMessage` instance holds the headers from an HTTP response. It is implemented using the *email.message.Message* class.

22.13 `ftplib` — FTP protocol client

Source code: Lib/ftplib.py

This module defines the class *FTP* and a few related items. The *FTP* class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module *urllib.request* to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet **RFC 959**.

Here's a sample session using the *ftplib* module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-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()
```

The module defines the following items:

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*)

Return a new instance of the *FTP* class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

The *FTP* class supports the `with` statement, e.g.:

```
>>> 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: Support for the `with` statement was added.

버전 3.3에서 변경: *source_address* parameter was added.

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *keyfile=None*, *certfile=None*, *context=None*, *timeout=None*, *source_address=None*)

A *FTP* subclass which adds TLS support to FTP as described in **RFC 4217**. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the *prot_p()* method. *context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read 보안 고려 사항 for best practices.

keyfile and *certfile* are a legacy alternative to *context* – they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

버전 3.2에 추가.

버전 3.3에서 변경: *source_address* parameter was added.

버전 3.4에서 변경: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

버전 3.6부터 폐지: *keyfile* and *certfile* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

Here's a sample session using the *FTP_TLS* class:

```
>>> 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']
```

exception ftplib.**error_reply**

Exception raised when an unexpected reply is received from the server.

exception ftplib.**error_temp**

Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.**error_perm**

Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.**error_proto**

Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.**all_errors**

The set of all exceptions (as a tuple) that methods of *FTP* instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as *OSError*.

더 보기:

Module ***netrc*** Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

22.13.1 FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

FTP instances have the following methods:

`FTP.`**`set_debuglevel`**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

`FTP.`**`connect`**(*host=''*, *port=0*, *timeout=None*, *source_address=None*)
Connect to the given host and port. The default port number is `21`, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If no *timeout* is passed, the global default timeout setting will be used. *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

버전 3.3에서 변경: *source_address* parameter was added.

`FTP.`**`getwelcome`**()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

`FTP.`**`login`**(*user='anonymous'*, *passwd=''*, *acct=''*)
Log in as the given *user*. The *passwd* and *acct* parameters are optional and default to the empty string. If no *user* is specified, it defaults to `'anonymous'`. If *user* is `'anonymous'`, the default *passwd* is `'anonymous@'`. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The *acct* parameter supplies "accounting information"; few systems implement this.

`FTP.`**`abort`**()
Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

`FTP.`**`sendcmd`**(*cmd*)
Send a simple command string to the server and return the response string.

`FTP.`**`voidcmd`**(*cmd*)
Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise *error_reply* otherwise.

`FTP.`**`retrbinary`**(*cmd*, *callback*, *blocksize=8192*, *rest=None*)
Retrieve a file in binary transfer mode. *cmd* should be an appropriate `RETR` command: `'RETR filename'`. The *callback* function is called for each block of data received, with a single bytes argument giving the data block. The optional *blocksize* argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to *callback*). A reasonable default is chosen. *rest* means the same thing as in the *transfercmd()* method.

`FTP.`**`retrlines`**(*cmd*, *callback=None*)
Retrieve a file or directory listing in ASCII transfer mode. *cmd* should be an appropriate `RETR` command (see *retrbinary()*) or a command such as `LIST` or `NLST` (usually just the string `'LIST'`). `LIST` retrieves a list of files and information about those files. `NLST` retrieves a list of file names. The *callback* function is called for

each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to `sys.stdout`.

`FTP.`**`set_pasv`**(*val*)
Enable "passive" mode if *val* is true, otherwise disable passive mode. Passive mode is on by default.

`FTP.`**`storbinary`**(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)
Store a file in binary transfer mode. *cmd* should be an appropriate `STOR` command: `"STOR filename"`. *fp* is a *file object* (opened in binary mode) which is read until EOF using its `read()` method in blocks of size *blocksize* to provide the data to be stored. The *blocksize* argument defaults to 8192. *callback* is an optional single parameter callable that is called on each block of data after it is sent. *rest* means the same thing as in the *transfercmd()* method.

버전 3.2에서 변경: *rest* parameter added.

`FTP.`**`storlines`**(*cmd*, *fp*, *callback=None*)
Store a file in ASCII transfer mode. *cmd* should be an appropriate `STOR` command (see *storbinary()*). Lines are read until EOF from the *file object* *fp* (opened in binary mode) using its *readline()* method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

`FTP.`**`transfercmd`**(*cmd*, *rest=None*)
Initiate a transfer over the data connection. If the transfer is active, send an `EPRT` or `PORT` command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an `EPSV` or `PASV` command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a `REST` command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that **RFC 959** requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The *transfercmd()* method, therefore, converts *rest* to a string, but no check is performed on the string's contents. If the server does not recognize the `REST` command, an *error_reply* exception will be raised. If this happens, simply call *transfercmd()* without a *rest* argument.

`FTP.`**`ntransfercmd`**(*cmd*, *rest=None*)
Like *transfercmd()*, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. *cmd* and *rest* means the same thing as in *transfercmd()*.

`FTP.`**`mlsd`**(*path=""*, *facts=[]*)
List a directory in a standardized format by using `MLSD` command (**RFC 3659**). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. `["type", "size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

버전 3.3에 추가.

`FTP.`**`nlst`**(*argument*[, ...])
Return a list of file names as returned by the `NLST` command. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command.

참고: If your server supports the command, *mlsd()* offers a better API.

`FTP.`**`dir`**(*argument*[, ...])
Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a *callback* function as for *retrlines()*; the default prints to `sys.stdout`. This method returns `None`.

참고: If your server supports the command, *mlsd()* offers a better API.

FTP.**rename**(*fromname*, *toname*)
Rename file *fromname* on the server to *toname*.

FTP.**delete**(*filename*)
Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise raises *error_perm* on permission errors or *error_reply* on other errors.

FTP.**cwd**(*pathname*)
Set the current directory on the server.

FTP.**mkd**(*pathname*)
Create a new directory on the server.

FTP.**pwd**()
Return the pathname of the current directory on the server.

FTP.**rmd**(*dirname*)
Remove the directory named *dirname* on the server.

FTP.**size**(*filename*)
Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

FTP.**quit**()
Send a `QUIT` command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the *close()* method which renders the *FTP* instance useless for subsequent calls (see below).

FTP.**close**()
Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to *quit()*. After this call the *FTP* instance should not be used any more (after a call to *close()* or *quit()* you cannot reopen the connection by issuing another *login()* method).

22.13.2 FTP_TLS Objects

FTP_TLS class inherits from *FTP*, defining these additional objects:

FTP_TLS.**ssl_version**
The SSL version to use (defaults to *ssl.PROTOCOL_SSLv23*).

FTP_TLS.**auth**()
Set up a secure control connection by using TLS or SSL, depending on what is specified in the *ssl_version* attribute.

버전 3.4에서 변경: The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

FTP_TLS.**ccc**()
Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

버전 3.3에 추가.

FTP_TLS.**prot_p**()
Set up secure data connection.

FTP_TLS.**prot_c**()
Set up clear text data connection.

22.14 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.

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).

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *keyfile=None*, *certfile=None*, *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.

*keyfile*과 *certfile*은 *context*의 레거시 대안입니다 - SSL 연결을 위해 각각 PEM 형식의 개인 키와 인증서 체인 파일을 가리킬 수 있습니다.

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 3.6부터 폐지: *keyfile*과 *certfile*은 폐지되었고, *context*로 대체합니다. 대신 *ssl.SSLContext.load_cert_chain()*을 사용하거나, *ssl.create_default_context()*가 시스템의 신뢰할 수 있는 CA 인증서를 선택하도록 하십시오.

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.

22.14.1 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.

버전 3.4에 추가.

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 QPOP인데, 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.

버전 3.5에 추가.

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)

버전 3.4에 추가.

*POP3_SSL*의 인스턴스에는 추가 메서드가 없습니다. 이 서브 클래스의 인터페이스는 그 부모와 같습니다.

22.14.2 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())
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.

22.15 imaplib — IMAP4 protocol client

Source code: Lib/imaplib.py

This module defines three classes, *IMAP4*, *IMAP4_SSL* and *IMAP4_stream*, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060**. It is backward compatible with IMAP4 (**RFC 1730**) servers, but note that the STATUS command is not supported in IMAP4.

Three classes are provided by the *imaplib* module, *IMAP4* is the base class:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*)

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used.

The *IMAP4* class supports the `with` statement. When used like this, the IMAP4 LOGOUT command is issued automatically when the `with` statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

버전 3.5에서 변경: Support for the `with` statement was added.

Three exceptions are defined as attributes of the *IMAP4* class:

exception IMAP4.**error**

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.**abort**

IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.**readonly**

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, *keyfile=None*, *certfile=None*, *ssl_context=None*)

This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read 보안 고려 사항 for best practices.

keyfile and *certfile* are a legacy alternative to *ssl_context* - they can point to PEM-formatted private key and certificate chain files for the SSL connection. Note that the *keyfile*/*certfile* parameters are mutually exclusive with *ssl_context*, a *ValueError* is raised if *keyfile*/*certfile* is provided along with *ssl_context*.

버전 3.3에서 변경: *ssl_context* parameter added.

버전 3.4에서 변경: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

버전 3.6부터 폐지: *keyfile* and *certfile* are deprecated in favor of *ssl_context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

The second subclass allows for connections created by a child process:

class imaplib.**IMAP4_stream**(*command*)
This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined:

imaplib.**Internaldate2tuple**(*datestr*)
Parse an IMAP4 `INTERNALDATE` string and return corresponding local time. The return value is a *time.struct_time* tuple or `None` if the string has wrong format.

imaplib.**Int2AP**(*num*)
Converts an integer into a string representation using characters from the set [`A` .. `P`].

imaplib.**ParseFlags**(*flagstr*)
Converts an IMAP4 `FLAGS` response to a tuple of individual flags.

imaplib.**Time2Internaldate**(*date_time*)
Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

더 보기:

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's *IMAP Information Center* (https://www.washington.edu/imap/).

22.15.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to `STORE`) then enclose the string in parentheses (eg: `r'(\Deleted)'`).

Each command returns 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 string, 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).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

An *IMAP4* instance has the following methods:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)
Append *message* to named mailbox.

IMAP4.**authenticate**(*mechanism*, *authobject*)
Authenticate command — requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

버전 3.5에서 변경: string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

IMAP4.**check**()
Checkpoint mailbox on server.

IMAP4.**close**()
Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

IMAP4.**copy**(*message_set*, *new_mailbox*)
Copy *message_set* messages onto end of *new_mailbox*.

IMAP4.**create**(*mailbox*)
Create new mailbox named *mailbox*.

IMAP4.**delete**(*mailbox*)
Delete old mailbox named *mailbox*.

IMAP4.**deleteacl**(*mailbox*, *who*)
Delete the ACLs (remove any rights) set for who on mailbox.

IMAP4.**enable**(*capability*)
Enable *capability* (see **RFC 5161**). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see **RFC 6855**).

버전 3.5에 추가: The *`enable()`* method itself, and **RFC 6855** support.

IMAP4.**expunge**()
Permanently remove deleted items from selected mailbox. Generates an `EXPUNGE` response for each deleted message. Returned data contains a list of `EXPUNGE` message numbers in order received.

IMAP4.**fetch**(*message_set*, *message_parts*)
Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: `"(UID BODY[TEXT])"`. Returned data are tuples of message part envelope and data.

IMAP4.**getacl**(*mailbox*)
Get the `ACL`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)
Retrieve the specified `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**getquota**(*root*)
Get the `quota` *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**getquotaroot**(*mailbox*)
Get the list of `quota roots` for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**list**([*directory*[, *pattern*]])
List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of LIST responses.

IMAP4.**login**(*user*, *password*)
Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.**login_cram_md5**(*user*, *password*)
Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

IMAP4.**logout**()
Shutdown connection to server. Returns server BYE response.

IMAP4.**lsub**(*directory=""*, *pattern='*'*)
List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.**myrights**(*mailbox*)
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

IMAP4.**namespace**()
Returns IMAP namespaces as defined in **RFC 2342**.

IMAP4.**noop**()
Send NOOP to server.

IMAP4.**open**(*host*, *port*)
Opens socket to *port* at *host*. This method is implicitly called by the *IMAP4* constructor. The connection objects established by this method will be used in the *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, and *IMAP4.shutdown()* methods. You may override this method.

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.**proxyauth**(*user*)
Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

IMAP4.**read**(*size*)
Reads *size* bytes from the remote server. You may override this method.

IMAP4.**readline**()
Reads one line from the remote server. You may override this method.

IMAP4.**recent**()
Prompt server for an update. Returned data is None if no new messages, else value of RECENT response.

IMAP4.**rename**(*oldmailbox*, *newmailbox*)
Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.**response**(*code*)
Return data for response *code* if received, or None. Returns the given code, instead of the usual type.

IMAP4.**search**(*charset*, *criterion*[, ...])
Search mailbox for matching messages. *charset* may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be None if the UTF8=ACCEPT capability was enabled using the *enable()* command.

Example:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)
Select a mailbox. Returned data is the count of messages in *mailbox* (EXISTS response). The default *mailbox* is 'INBOX'. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.**send**(*data*)
Sends data to the remote server. You may override this method.

IMAP4.**setacl**(*mailbox*, *who*, *what*)
Set an ACL for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])
Set ANNOTATIONs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**setquota**(*root*, *limits*)
Set the quota *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**shutdown**()
Close connection established in open. This method is implicitly called by *IMAP4.logout()*. You may override this method.

IMAP4.**socket**()
Returns socket instance used to connect to server.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])
The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

IMAP4.**starttls**(*ssl_context=None*)
Send a STARTTLS command. The *ssl_context* argument is optional and should be a *ssl.SSLContext* object. This will enable encryption on the IMAP connection. Please read 보안 고려 사항 for best practices.

버전 3.2에 추가.

버전 3.4에서 변경: The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

IMAP4.**status**(*mailbox*, *names*)
Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)
Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

참고: Creating flags containing ']' (for example: "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib nonetheless continues to allow such tags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.**subscribe**(*mailbox*)
Subscribe to new mailbox.

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])
The thread command is a variant of search with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid thread command which corresponds to thread the way that uid search corresponds to search. The thread command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an IMAP4rev1 extension command.

IMAP4.**uid**(*command*, *arg*[, ...])
Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

IMAP4.**unsubscribe**(*mailbox*)
Unsubscribe from old mailbox.

IMAP4.**xatom**(*name*[, ...])
Allow simple extension commands notified by server in CAPABILITY response.

The following attributes are defined on instances of *IMAP4*:

IMAP4.**PROTOCOL_VERSION**
The most recent supported protocol in the CAPABILITY response from the server.

IMAP4.**debug**
Integer value to control debugging output. The initialize value is taken from the module variable Debug. Values greater than three trace each command.

IMAP4.**utf8_enabled**
Boolean value that is normally False, but is set to True if an *enable()* command is successfully issued for the UTF8=ACCEPT capability.

버전 3.5에 추가.

22.15.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4()
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()
```

22.16 nntplib — NNTP protocol client

Source code: Lib/nntplib.py

This module defines the class *NNTP* which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with **RFC 3977** as well as the older **RFC 977** and **RFC 2980**.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```

To post an article from a binary file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> f = open('article.txt', 'rb')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

The module itself defines the following classes:

class nntplib.**NNTP**(*host*, *port=119*, *user=None*, *password=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP* object, representing a connection to the NNTP server running on host *host*, listening at port *port*. An optional *timeout* can be specified for the socket connection. If the optional *user* and *password* are provided, or if suitable credentials are present in `/.netrc` and the optional flag *usenetrc* is true, the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag *readermode* is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected *NNTPPermanentError*s, you might need to set *readermode*. The *NNTP* class supports the `with` statement to unconditionally consume *OSError* exceptions and to close the NNTP connection when done, e.g.:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.io') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.
↪python.committers')
>>>
```

버전 3.2에서 변경: *usenetrc* is now `False` by default.

버전 3.3에서 변경: Support for the `with` statement was added.

class nntplib.**NNTP_SSL**(*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP_SSL* object, representing an encrypted connection to the NNTP server running on host *host*, listening at port *port*. *NNTP_SSL* objects have the same methods as *NNTP* objects. If *port* is omitted, port 563 (NNTPS) is used. *ssl_context* is also optional, and is a *SSLContext* object. Please read 보안 고려 사항 for best practices. All other parameters behave the same as for *NNTP*.

Note that SSL-on-563 is discouraged per **RFC 4642**, in favor of STARTTLS as described below. However, some servers only support the former.

버전 3.2에 추가.

버전 3.4에서 변경: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

exception nntplib.**NNTPError**

Derived from the standard exception *Exception*, this is the base class for all exceptions raised by the *nntplib* module. Instances of this class have the following attribute:

response

The response of the server if available, as a *str* object.

exception nntplib.**NNTPReplyError**

Exception raised when an unexpected reply is received from the server.

exception nntplib.**NNTPTemporaryError**

Exception raised when a response code in the range 400–499 is received.

exception nntplib.**NNTPPermanentError**
Exception raised when a response code in the range 500–599 is received.

exception nntplib.**NNTPProtocolError**
Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

exception nntplib.**NNTPDataError**
Exception raised when there is some error in the response data.

22.16.1 NNTP Objects

When connected, *NNTP* and *NNTP_SSL* objects support the following methods and attributes.

Attributes

NNTP.**nntp_version**
An integer representing the version of the NNTP protocol supported by the server. In practice, this should be 2 for servers advertising **RFC 3977** compliance and 1 for others.

버전 3.2에 추가.

NNTP.**nntp_implementation**
A string describing the software name and version of the NNTP server, or *None* if not advertised by the server.

버전 3.2에 추가.

Methods

The *response* that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

Many of the following methods take an optional keyword-only argument *file*. When the *file* argument is supplied, it must be either a *file object* opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the server (except for the response line and the terminating dot) to the file; any list of lines, tuples or objects that the method normally returns will be empty.

버전 3.2에서 변경: Many of the following methods have been reworked and fixed, which makes them incompatible with their 3.1 counterparts.

NNTP.**quit**()
Send a QUIT command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

NNTP.**getwelcome**()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

NNTP.**getcapabilities**()
Return the **RFC 3977** capabilities advertised by the server, as a *dict* instance mapping capability names to (possibly empty) lists of values. On legacy servers which don't understand the CAPABILITIES command, an empty dictionary is returned instead.

```
>>> s = NNTP('news.gmane.io')
>>> 'POST' in s.getcapabilities()
True
```

버전 3.2에 추가.

NNTP.**login**(*user=None*, *password=None*, *usenetrc=True*)
Send AUTHINFO commands with the user name and password. If *user* and *password* are None and *usenetrc* is true, credentials from ~/.netrc will be used if possible.

Unless intentionally delayed, login is normally performed during the *NNTP* object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set *user* or *password* when creating the object, and must set *usenetrc* to False.

버전 3.2에 추가.

NNTP.**starttls**(*context=None*)
Send a STARTTLS command. This will enable encryption on the NNTP connection. The *context* argument is optional and should be a *ssl.SSLContext* object. Please read 보안 고려 사항 for best practices.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a *NNTP* object initialization. See *NNTP.login()* for information on suppressing this behavior.

버전 3.2에 추가.

버전 3.4에서 변경: The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

NNTP.**newgroups**(*date*, *, *file=None*)
Send a NEWGROUPS command. The *date* argument should be a *datetime.date* or *datetime.datetime* object. Return a pair (response, groups) where *groups* is a list representing the groups that are new since the given *date*. If *file* is supplied, though, then *groups* will be empty.

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP.**newnews**(*group*, *date*, *, *file=None*)
Send a NEWNEWS command. Here, *group* is a group name or '*', and *date* has the same meaning as for *newgroups()*. Return a pair (response, articles) where *articles* is a list of message ids.

This command is frequently disabled by NNTP server administrators.

NNTP.**list**(*group_pattern=None*, *, *file=None*)
Send a LIST or LIST ACTIVE command. Return a pair (response, list) where *list* is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string *group_pattern*. Each tuple has the form (group, last, first, flag), where *group* is a group name, *last* and *first* are the last and first article numbers, and *flag* usually takes one of these values:

- y: Local postings and articles from peers are allowed.
- m: The group is moderated and all postings must be approved.
- n: No local postings are allowed, only articles from peers.
- j: Articles from peers are filed in the junk group instead.
- x: No local postings, and articles from peers are ignored.
- =foo.bar: Articles are filed in the foo.bar group instead.

If *flag* has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if *group_pattern* is not specified. It is best to cache the results offline unless you really need to refresh them.

버전 3.2에서 변경: *group_pattern* was added.

NNTP.**descriptions**(*grouppattern*)
Send a `LIST NEWSGROUPS` command, where *grouppattern* is a wildmat string as specified in **RFC 3977** (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair `(response, descriptions)`, where *descriptions* is a dictionary mapping group names to textual descriptions.

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP.**description**(*group*)
Get a description for a single group *group*. If more than one group matches (if 'group' is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use *descriptions()*.

NNTP.**group**(*name*)
Send a `GROUP` command, where *name* is the group name. The group is selected as the current group, if it exists. Return a tuple `(response, count, first, last, name)` where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name.

NNTP.**over**(*message_spec*, *, *file=None*)
Send an `OVER` command, or an `XOVER` command on legacy servers. *message_spec* can be either a string representing a message id, or a `(first, last)` tuple of numbers indicating a range of articles in the current group, or a `(first, None)` tuple indicating a range of articles starting from *first* to the last article in the current group, or *None* to select the current article in the current group.

Return a pair `(response, overviews)`. *overviews* is a list of `(article_number, overview)` tuples, one for each article selected by *message_spec*. Each *overview* is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with `":"`). The following items are guaranteed to be present by the NNTP specification:

- the `subject`, `from`, `date`, `message-id` and `references` headers
- the `:bytes` metadata: the number of bytes in the entire raw article (including headers and body)
- the `:lines` metadata: the number of lines in the article body

The value of each item is either a string, or *None* if not present.

It is advisable to use the *decode_header()* function on header values when they may contain non-ASCII characters:

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject
↪']
>>> over['from']
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

버전 3.2에 추가.

NNTP.**help**(*, *file=None*)
Send a HELP command. Return a pair `(response, list)` where *list* is a list of help strings.

NNTP.**stat**(*message_spec=None*)
Send a STAT command, where *message_spec* is either a message id (enclosed in '<' and '>') or an article number in the current group. If *message_spec* is omitted or *None*, the current article in the current group is considered. Return a triple `(response, number, id)` where *number* is the article number and *id* is the message id.

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next**()
Send a NEXT command. Return as for *stat()*.

NNTP.**last**()
Send a LAST command. Return as for *stat()*.

NNTP.**article**(*message_spec=None*, *, *file=None*)
Send an ARTICLE command, where *message_spec* has the same meaning as for *stat()*. Return a tuple `(response, info)` where *info* is a *namedtuple* with three attributes *number*, *message_id* and *lines* (in that order). *number* is the article number in the group (or 0 if the information is not available), *message_id* the message id as a string, and *lines* a list of lines (without terminating newlines) comprising the raw message including headers and body.

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head**(*message_spec=None*, *, *file=None*)
Same as *article()*, but sends a HEAD command. The *lines* returned (or written to *file*) will only contain the message headers, not the body.

NNTP.**body**(*message_spec=None*, *, *file=None*)
Same as *article()*, but sends a BODY command. The *lines* returned (or written to *file*) will only contain the message body, not the headers.

NNTP.**post**(*data*)
Post an article using the POST command. The *data* argument is either a *file object* opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed

news article, including the required headers. The *post()* method automatically escapes lines beginning with . and appends the termination line.

If the method succeeds, the server's response is returned. If the server refuses posting, a *NNTPReplyError* is raised.

NNTP.**ihave**(*message_id*, *data*)
Send an IHAVE command. *message_id* is the id of the message to send to the server (enclosed in '<' and '>'). The *data* parameter and the return value are the same as for *post()*.

NNTP.**date**()
Return a pair (response, date). *date* is a *datetime* object containing the current date and time of the server.

NNTP.**slave**()
Send a SLAVE command. Return the server's *response*.

NNTP.**set_debuglevel**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

The following are optional NNTP extensions defined in **RFC 2980**. Some of them have been superseded by newer commands in **RFC 3977**.

NNTP.**xhdr**(*hdr*, *str*, *, *file=None*)
Send an XHDR command. The *hdr* argument is a header keyword, e.g. 'subject'. The *str* argument should have the form 'first-last' where *first* and *last* are the first and last article numbers to search. Return a pair (response, list), where *list* is a list of pairs (id, text), where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the XHDR command is stored in a file. If *file* is a string, then the method will open a file with that name, write to it then close it. If *file* is a *file object*, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**xover**(*start*, *end*, *, *file=None*)
Send an XOVER command. *start* and *end* are article numbers delimiting the range of articles to select. The return value is the same of for *over()*. It is recommended to use *over()* instead, since it will automatically use the newer OVER command if available.

NNTP.**xpath**(*id*)
Return a pair (resp, path), where *path* is the directory path to the article with message ID *id*. Most of the time, this extension is not enabled by NNTP server administrators.

버전 3.3부터 폐지: The XPATH extension is not actively used.

22.16.2 Utility functions

The module also defines the following utility function:

nntplib.**decode_header**(*header_str*)
Decode a header value, un-escaping any escaped non-ASCII characters. *header_str* must be a *str* object. The unescaped value is returned. Using this function is recommended to display some headers in a human readable form:

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

22.17 smtplib — SMTP protocol client

Source code: Lib/smtplib.py

The *smtplib* module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** (Simple Mail Transfer Protocol) and **RFC 1869** (SMTP Service Extensions).

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*[, *timeout*], *source_address=None*)

An *SMTP* instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP *connect()* method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using *socket.getfqdn()*. If the *connect()* call returns anything other than a success code, an *SMTPConnectError* is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, *socket.timeout* is raised. The optional source_address parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple (host, port), for the socket to bind to as its source address before connecting. If omitted (or if host or port are `''` and/or 0 respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, *sendmail()*, and *SMTP.quit()* methods. An example is included below.

The *SMTP* class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

버전 3.3에서 변경: Support for the `with` statement was added.

버전 3.3에서 변경: source_address argument was added.

버전 3.5에 추가: The SMTPUTF8 extension (**RFC 6531**) is now supported.

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *keyfile=None*, *certfile=None*[, *timeout*], *context=None*, *source_address=None*)

An *SMTP_SSL* instance behaves exactly the same as instances of *SMTP*. *SMTP_SSL* should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have the same meaning as they do in the *SMTP* class. *context*, also optional, can contain a *SSLContext* and allows configuring various aspects of the secure connection. Please read 보안 고려 사항 for best practices.

keyfile and *certfile* are a legacy alternative to *context*, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

버전 3.3에서 변경: *context* was added.

버전 3.3에서 변경: source_address argument was added.

버전 3.4에서 변경: The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

버전 3.6부터 폐지: *keyfile* and *certfile* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*)

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments local_hostname and source_address have the same meaning as they do in the `SMTP` class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

A nice selection of exceptions is defined as well:

exception smtplib.**SMTPException**

Subclass of `OSError` that is the base exception class for all the other exceptions provided by this module.

버전 3.4에서 변경: SMTPException became subclass of `OSError`

exception smtplib.**SMTPServerDisconnected**

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the `SMTP` instance before connecting it to a server.

exception smtplib.**SMTPResponseException**

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.**SMTPSenderRefused**

Sender address refused. In addition to the attributes set by on all `SMTPResponseException` exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.**SMTPRecipientsRefused**

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as `SMTP.sendmail()` returns.

exception smtplib.**SMTPDataError**

The SMTP server refused to accept the message data.

exception smtplib.**SMTPConnectError**

Error occurred during establishment of a connection with the server.

exception smtplib.**SMTPHeloError**

The server refused our `HELO` message.

exception smtplib.**SMTPNotSupportedError**

The command or option attempted is not supported by the server.

버전 3.5에 추가.

exception smtplib.**SMTPAuthenticationError**
SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

더 보기:

RFC 821 - Simple Mail Transfer Protocol Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

22.17.1 SMTP Objects

An *SMTP* instance has the following methods:

SMTP.**set_debuglevel**(*level*)
Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

버전 3.5에서 변경: Added debuglevel 2.

SMTP.**docmd**(*cmd*, *args=''*)
Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.**connect**(*host='localhost'*, *port=0*)
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (`':'`) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

SMTP.**helo**(*name=''*)
Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.**ehlo**(*name=''*)
Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by *has_extn()*. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to true or false depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.**ehlo_or_helo_if_needed**()
This method calls *ehlo()* and/or *helo()* if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTP.**has_extn**(*name*)
Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.**verify**(*address*)
Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

참고: Many sites disable SMTP VRFY in order to foil spammers.

SMTP.**login**(*user*, *password*, *, *initial_response_ok=True*)
Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError The server didn't accept the username/password combination.

SMTPNotSupportedError The AUTH command is not supported by the server.

SMTPException No suitable authentication method was found.

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** can be sent along with the AUTH command, rather than requiring a challenge/response.

버전 3.5에서 변경: *SMTPNotSupportedError* may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)
Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the auth element of esmtp_features.

authobject must be a callable object taking an optional single argument:

data = authobject(challenge=None)

If optional keyword argument *initial_response_ok* is true, authobject() will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII str which will be encoded and sent with the AUTH command as below. If the authobject() does not support an initial response (e.g. because it requires a challenge), it should return None when called with challenge=None. If *initial_response_ok* is false, then authobject() will not be called first with None.

If the initial response check returns `None`, or if *initial_response_ok* is false, `authobject()` will be called to process the server's challenge response; the *challenge* argument it is passed will be a `bytes`. It should return ASCII `str` *data* that will be base64 encoded and sent to the server.

The `SMTP` class provides `authobjects` for the `CRAM-MD5`, `PLAIN`, and `LOGIN` mechanisms; they are named `SMTP.auth_cram_md5`, `SMTP.auth_plain`, and `SMTP.auth_login` respectively. They all require that the `user` and `password` properties of the `SMTP` instance are set to appropriate values.

User code does not normally need to call `auth` directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. `auth` is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

버전 3.5에 추가.

SMTP.**starttls**(*keyfile=None*, *certfile=None*, *context=None*)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be `None`.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first.

버전 3.6부터 폐지: *keyfile* and *certfile* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

SMTPHeloError The server didn't reply properly to the `HELO` greeting.

SMTPNotSupportedError The server does not support the STARTTLS extension.

RuntimeError SSL/TLS support is not available to your Python interpreter.

버전 3.3에서 변경: *context* was added.

버전 3.4에서 변경: The method now supports hostname check with `SSLContext.check_hostname` and *Server Name Indicator* (see *HAS_SNI*).

버전 3.5에서 변경: The error raised for lack of STARTTLS support is now the *SMTPNotSupportedError* subclass instead of the base *SMTPException*.

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

Send mail. The required arguments are an **RFC 822** from-address string, a list of **RFC 822** to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as `8bitmime`) to be used in `MAIL FROM` commands as *mail_options*. ESMTP options (such as `DSN` commands) that should be used with all `RCPT` commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as `mail()`, `rcpt()` and `data()` to send the message.)

참고: The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

---

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions:

SMTPRecipientsRefused All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError The server didn't reply properly to the `HELO` greeting.

SMTPSenderRefused The server didn't accept the *from_addr*.

SMTPDataError The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError `SMTPUTF8` was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

버전 3.2에서 변경: *msg* may be a byte string.

버전 3.5에서 변경: `SMTPUTF8` support added, and *SMTPNotSupportedError* may be raised if `SMTPUTF8` is specified but the server does not support it.

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

This is a convenience method for calling *sendmail()* with the message represented by an *email.message.Message* object. The arguments have the same meaning as for *sendmail()*, except that *msg* is a `Message` object.

If *from_addr* is `None` or *to_addrs* is `None`, `send_message` fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322**: *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a *ValueError* is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

`send_message` serializes *msg* using *BytesGenerator* with `\r\n` as the *linesep*, and calls *sendmail()* to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, `send_message` does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise `SMTPUTF8` support, an `SMTPNotSupported` error is raised. Otherwise the `Message` is serialized with a clone of its *policy* with the *utf8* attribute set to `True`, and `SMTPUTF8` and `BODY=8BITMIME` are added to *mail_options*.

버전 3.2에 추가.

버전 3.5에 추가: Support for internationalized addresses (`SMTPUTF8`).

SMTP.**quit**()

Terminate the SMTP session and close the connection. Return the result of the SMTP `QUIT` command.

Low-level methods corresponding to the standard SMTP/ESMTP commands `HELP`, `RSET`, `NOOP`, `MAIL`, `RCPT`, and `DATA` are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

22.17.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly.

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

참고: In general, you will want to use the *email* package's features to construct an email message, which you can then send via *send_message()*; see *email:* 예제.

22.18 smtpd — SMTP Server

Source code: Lib/smtpd.py

This module offers several classes to implement SMTP (email) servers.

더 보기:

The aiosmtpd package is a recommended replacement for this module. It is based on *asyncio* and provides a more straightforward API. *smtpd* should be considered deprecated.

Several server implementations are present; one is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

Additionally the SMTPChannel may be extended to implement very specific interaction behaviour with SMTP clients.

The code supports **RFC 5321**, plus the **RFC 1870** SIZE and **RFC 6531** SMTPUTF8 extensions.

22.18.1 SMTPServer Objects

class smtpd.**SMTPServer**(*localaddr*, *remoteaddr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

Create a new *SMTPServer* object, which binds to local address *localaddr*. It will treat *remoteaddr* as an upstream SMTP relayer. Both *localaddr* and *remoteaddr* should be a *(host, port)* tuple. The object inherits from *asyncore.dispatcher*, and so will insert itself into *asyncore*'s event loop on instantiation.

data_size_limit specifies the maximum number of bytes that will be accepted in a DATA command. A value of None or 0 means no limit.

map is the socket map to use for connections (an initially empty dictionary is a suitable value). If not specified the *asyncore* global socket map is used.

enable_SMTPUTF8 determines whether the SMTPUTF8 extension (as defined in **RFC 6531**) should be enabled. The default is False. When True, SMTPUTF8 is accepted as a parameter to the MAIL command and when present is passed to *process_message()* in the kwargs['mail_options'] list. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. When *decode_data* is False (the default), the server advertises the 8BITMIME extension (**RFC 6152**), accepts the BODY=8BITMIME parameter to the MAIL command, and when present passes it to *process_message()* in the kwargs['mail_options'] list. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

process_message(*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)

Raise a *NotImplementedError* exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as *remoteaddr* will be available as the _remoteaddr attribute. *peer* is the remote host's address, *mailfrom* is the envelope originator, *rcpttos* are the envelope recipients and *data* is a string containing the contents of the e-mail (which should be in **RFC 5321** format).

If the *decode_data* constructor keyword is set to True, the *data* argument will be a unicode string. If it is set to False, it will be a bytes object.

kwargs is a dictionary containing additional information. It is empty if decode_data=True was given as an init argument, otherwise it contains the following keys:

***mail_options*:** a list of all received parameters to the MAIL command (the elements are uppercase strings; example: ['BODY=8BITMIME', 'SMTPUTF8']).

***rcpt_options*:** same as *mail_options* but for the RCPT command. Currently no RCPT TO options are supported, so for now this will always be an empty list.

Implementations of process_message should use the **kwargs signature to accept arbitrary keyword arguments, since future feature enhancements may add keys to the kwargs dictionary.

Return None to request a normal 250 Ok response; otherwise return the desired response string in **RFC 5321** format.

channel_class

Override this in subclasses to use a custom *SMTPChannel* for managing SMTP clients.

버전 3.4에 추가: The *map* constructor argument.

버전 3.5에서 변경: *localaddr* and *remoteaddr* may now contain IPv6 addresses.

버전 3.5에 추가: The *decode_data* and *enable_SMTPUTF8* constructor parameters, and the *kwargs* parameter to *process_message()* when *decode_data* is False.

버전 3.6에서 변경: *decode_data* is now `False` by default.

22.18.2 DebuggingServer Objects

class smtpd.**DebuggingServer**(*localaddr*, *remoteaddr*)
Create a new debugging server. Arguments are as per *SMTPServer*. Messages will be discarded, and printed on stdout.

22.18.3 PureProxy Objects

class smtpd.**PureProxy**(*localaddr*, *remoteaddr*)
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*. Note that running this has a good chance to make you into an open relay, so please be careful.

22.18.4 MailmanProxy Objects

class smtpd.**MailmanProxy**(*localaddr*, *remoteaddr*)
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

22.18.5 SMTPChannel Objects

class smtpd.**SMTPChannel**(*server*, *conn*, *addr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)
Create a new *SMTPChannel* object which manages the communication between the server and a single SMTP client.

conn and *addr* are as per the instance variables described below.

data_size_limit specifies the maximum number of bytes that will be accepted in a `DATA` command. A value of `None` or `0` means no limit.

enable_SMTPUTF8 determines whether the `SMTPUTF8` extension (as defined in **RFC 6531**) should be enabled. The default is `False`. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

A dictionary can be specified in *map* to avoid using a global socket map.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. The default is `False`. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

To use a custom SMTPChannel implementation you need to override the *SMTPServer.channel_class* of your *SMTPServer*.

버전 3.5에서 변경: The *decode_data* and *enable_SMTPUTF8* parameters were added.

버전 3.6에서 변경: *decode_data* is now `False` by default.

The *SMTPChannel* has the following instance variables:

smtp_server
Holds the *SMTPServer* that spawned this channel.

conn
Holds the socket object connecting to the client.

addr
Holds the address of the client, the second value returned by *socket.accept*

received_lines
Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their `"\r\n"` line ending translated to `"\n"`.

smtp_state
Holds the current state of the channel. This will be either `COMMAND` initially and then `DATA` after the client sends a "DATA" line.

seen_greeting
Holds a string containing the greeting sent by the client in its "HELO".

mailfrom
Holds a string containing the address identified in the "MAIL FROM:" line from the client.

rcpttos
Holds a list of strings containing the addresses identified in the "RCPT TO:" lines from the client.

received_data
Holds a string containing all of the data sent by the client during the DATA state, up to but not including the terminating `"\r\n.\r\n"`.

fqdn
Holds the fully-qualified domain name of the server as returned by *socket.getfqdn()*.

peer
Holds the name of the client peer as returned by `conn.getpeername()` where `conn` is *conn*.

The *SMTPChannel* operates by invoking methods named `smtp_<command>` upon reception of a command line from the client. Built into the base *SMTPChannel* class are methods for handling the following commands (and responding to them appropriately):

Command	Action taken
HELO	Accepts the greeting from the client and stores it in *seen_greeting*. Sets server to base command mode.
EHLO	Accepts the greeting from the client and stores it in *seen_greeting*. Sets server to extended command mode.
NOOP	Takes no action.
QUIT	Closes the connection cleanly.
MAIL	Accepts the "MAIL FROM:" syntax and stores the supplied address as *mailfrom*. In extended command mode, accepts the **RFC 1870** SIZE attribute and responds appropriately based on the value of *data_size_limit*.
RCPT	Accepts the "RCPT TO:" syntax and stores the supplied addresses in the *rcpttos* list.
RSET	Resets the *mailfrom*, *rcpttos*, and *received_data*, but not the greeting.
DATA	Sets the internal state to `DATA` and stores remaining lines from the client in *received_data* until the terminator `"\r\n.\r\n"` is received.
HELP	Returns minimal information on command syntax
VRFY	Returns code 252 (the server doesn't know if the address is valid)
EXPN	Reports that the command is not implemented.

22.19 telnetlib — 텔넷 클라이언트

소스 코드: Lib/telnetlib.py

---

telnetlib 모듈은 텔넷 프로토콜을 구현하는 *Telnet* 클래스를 제공합니다. 프로토콜에 대한 자세한 내용은 **RFC 854**를 참조하십시오. 또한, 프로토콜 문자(아래를 보십시오)와 텔넷 옵션을 위한 기호 상수를 제공합니다. 텔넷 옵션의 기호 이름은 `arpa/telnet.h`의 정의를 따르며, 선행 `TELOPT_`는 제거됩니다. 전통적으로 `arpa/telnet.h`에 포함되지 않는 옵션의 기호 이름에 대해서는 모듈 소스 자체를 참조하십시오.

텔넷 명령을 위한 기호 상수는 다음과 같습니다: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class telnetlib.**Telnet**(*host=None*, *port=0*[, *timeout*])

Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the *open()* method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor too, in which case the connection to the server will be established before the constructor returns. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

이미 연결된 인스턴스를 다시 열지 마십시오.

이 클래스에는 많은 `read_*()` 메서드가 있습니다. 이들 중 일부는 연결의 끝을 읽을 때 *EOFError*를 발생시킴에 유의하십시오. 다른 이유로 빈 문자열을 반환할 수 있기 때문입니다. 아래의 개별 설명을 참조하십시오.

Telnet 객체는 컨텍스트 관리자이며 `with` 문에서 사용할 수 있습니다. `with` 블록이 끝날 때, *close()* 메서드가 호출됩니다:

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...
```

버전 3.6에서 변경: 컨텍스트 관리자 지원을 추가했습니다

더 보기:

RFC 854 - **Telnet Protocol Specification** 텔넷 프로토콜의 정의.

22.19.1 텔넷 객체

Telnet 인스턴스에는 다음과 같은 메서드가 있습니다.:

Telnet.**read_until**(*expected*, *timeout=None*)

주어진 바이트열 *expected*를 만나거나 *timeout* 초가 경과 할 때까지 읽습니다.

일치하는 것을 찾을 수 없으면, 사용 가능한 것을 대신 반환합니다. 빈 바이트열도 가능합니다. 연결이 닫혀 있고 사용할 수 있는 요리된 데이터가 없으면 *EOFError*를 발생시킵니다.

Telnet.**read_all**()

EOF까지 모든 데이터를 바이트열로 읽습니다; 연결이 닫힐 때까지 블록합니다.

Telnet.**read_some**()

EOF를 만나지 않으면 적어도 1바이트의 요리된 데이터를 읽습니다. EOF를 만나면 `b''`를 반환합니다. 즉시 사용할 수 있는 데이터가 없으면 블록합니다.

Telnet.**read_very_eager**()
I/O에서 블록하지 않고 읽을 수 있는 모든 것을 읽습니다 (eager).

연결이 닫혀 있고 사용할 수 있는 요리된 데이터가 없으면 *EOFError*를 발생시킵니다. 그렇지 않고 사용할 수 있는 요리된 데이터가 없으면 `b''`를 반환합니다. IAC 시퀀스의 중간에 있지 않으면 블록하지 않습니다.

Telnet.**read_eager**()
쉽게 사용할 수 있는 데이터를 읽습니다.

연결이 닫혀 있고 사용할 수 있는 요리된 데이터가 없으면 *EOFError*를 발생시킵니다. 그렇지 않고 사용할 수 있는 요리된 데이터가 없으면 `b''`를 반환합니다. IAC 시퀀스의 중간에 있지 않으면 블록하지 않습니다.

Telnet.**read_lazy**()
이미 큐에 있는 데이터를 처리하고 반환합니다 (lazy).

연결이 닫혀 있고 사용할 수 있는 데이터가 없으면 *EOFError*를 발생시킵니다. 그렇지 않고 사용할 수 있는 요리된 데이터가 없으면 `b''`를 반환합니다. IAC 시퀀스의 중간에 있지 않으면 블록하지 않습니다.

Telnet.**read_very_lazy**()
요리된 큐에 있는 모든 데이터를 반환합니다 (very lazy).

연결이 닫혀 있고 사용할 수 있는 데이터가 없으면 *EOFError*를 발생시킵니다. 그렇지 않고 사용할 수 있는 요리된 데이터가 없으면 `b''`를 반환합니다. 이 메서드는 절대 블록하지 않습니다.

Telnet.**read_sb_data**()
SB/SE 쌍 (suboption begin/end) 간에 수집된 데이터를 반환합니다. `SE` 명령으로 호출되었을 때 콜백은 이 데이터에 액세스해야 합니다. 이 방법은 절대 블록하지 않습니다.

Telnet.**open**(*host*, *port=0*[, *timeout*])
호스트에 연결합니다. 선택적 두 번째 인자는 포트 번호이며, 기본값은 표준 텔넷 포트 (23) 입니다. 선택적 *timeout* 매개 변수는 연결 시도와 같은 블로킹 연산에 대한 시간제한을 초로 지정합니다 (지정하지 않으면, 전역 기본 시간제한 설정이 사용됩니다).

이미 연결된 인스턴스를 다시 열려고 하지 마십시오.

Telnet.**msg**(*msg*, **args*)
디버그 수준이 > 0 일 때 디버그 메시지를 인쇄합니다. 추가 인자가 있으면, 표준 문자열 포매팅 연산자를 사용하여 메시지에 치환됩니다.

Telnet.**set_debuglevel**(*debuglevel*)
디버그 수준을 설정합니다. *debuglevel*의 값이 클수록, 더 많은 디버그 출력을 얻을 수 있습니다 (`sys.stdout`으로).

Telnet.**close**()
연결을 닫습니다.

Telnet.**get_socket**()
내부적으로 사용되는 소켓 객체를 반환합니다.

Telnet.**fileno**()
내부적으로 사용되는 소켓 객체의 파일 기술자를 반환합니다.

Telnet.**write**(*buffer*)
IAC 문자를 중복 (doubling) 해서 소켓에 바이트열을 기록합니다. 연결이 블록 되면 블록 할 수 있습니다. 연결이 닫히면 *OSError*가 발생할 수 있습니다.

버전 3.3에서 변경: 이 메서드는 방법은 *socket.error*를 발생시켰습니다. 이제는 *OSError*의 별칭입니다.

Telnet.**interact**()
상호 작용 함수, 매우 단순한 텔넷 클라이언트를 에뮬레이션합니다.

Telnet.**mt_interact**()
*interact()*의 다중 스레드 버전.

Telnet.**expect**(*list*, *timeout=None*)
정규식 리스트 중 하나가 일치할 때까지 읽습니다.

첫 번째 인자는 컴파일되었거나 (정규식 객체) 컴파일되지 않은 (바이트열) 정규식의 리스트입니다. 선택적 두 번째 인자는 초 단위의 시간제한입니다; 기본값은 무기한 블록 하는 것입니다.

세 항목의 튜플을 반환합니다: 일치하는 첫 번째 정규식의 리스트 인덱스; 반환된 일치 객체; 그리고 일치를 포함해서 그때까지 읽은 바이트열.

파일의 끝이 발견되고 아무런 바이트도 읽히지 않았으면, *EOFError*를 발생시킵니다. 그렇지 않으면, 아무것도 일치하지 않을 때, `(-1, None, data)`를 반환합니다. 여기서 *data*는 지금까지 받은 바이트열입니다 (시간 초과가 발생하면 빈 바이트열일 수 있습니다).

정규식이 탐욕적인 일치 (가령 `.*`)로 끝나거나 둘 이상의 정규식이 같은 입력과 일치 할 수 있으면, 결과는 비결정적이며, I/O 타이밍에 따라 달라질 수 있습니다.

Telnet.**set_option_negotiation_callback**(*callback*)
입력 흐름에서 텔넷 옵션을 읽을 때마다, 이 *callback*(설정되었다면)은 다음과 같은 매개 변수로 호출됩니다: callback(telnet socket, command (DO/DONT/WILL/WONT), option). telnetlib은 나중에 다른 작업을 수행하지 않습니다.

22.19.2 텔넷 예제

일반적인 사용을 보여주는 간단한 예제:

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

22.20 uuid — RFC 4122 에 따른 UUID 객체

소스 코드: Lib/uuid.py

이 모듈은 **RFC 4122**에서 명시한 버전 1, 3, 4 및 5의 UUID를 생성하기 위해 불변 *UUID* 객체(*UUID* 클래스)와 *uuid1()*, *uuid3()*, *uuid4()*, *uuid5()*를 제공합니다.

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**
버전 3.7에 추가.

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)
```

hex, *bytes*, *bytes_le*, *fields*, *int* 중 하나는 반드시 주어져야 합니다. *version* 인자는 선택사항입니다; 인자로 주어질 경우, 결과로 생성된 UUID는 **RFC 4122**에 따라 변종 및 버전 번호를 설정하고 주어진 *hex*, *bytes*, *bytes_le*, *fields*, *int* 비트를 오버라이딩 합니다.

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
time_low	UUID의 처음 32bit
time_mid	UUID의 다음 16bit
time_hi_version	UUID의 다음 16bit
clock_seq_hi_variant	UUID의 다음 8bit
clock_seq_low	UUID의 다음 8bit
node	UUID의 마지막 48bit
time	60bit 타임스탬프
clock_seq	14bit 시퀀스 번호

UUID.**hex**
16진수 32자리 문자열 UUID

UUID.**int**
128bit 정수 UUID

UUID.**urn**
RFC 4122에서 명시한 URN의 UUID

UUID.**variant**
UUID 변종은 UUID의 내부 레이아웃을 결정합니다. 이는 상수 *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, 및 *RESERVED_FUTURE* 중 하나입니다.

UUID.**version**
UUID 버전 번호 (1부터 5까지이며, 변종이 *RFC_4122*일 때만 유효)

UUID.**is_safe**
플랫폼이 UUID를 다중 프로세스에 안전한 방식으로 생성했는지를 나타내는 *SafeUUID*의 열거체입니다.

버전 3.7에 추가.

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*)
호스트 ID, 시퀀스 번호 및 현재 시각으로 UUID 생성. *node*가 주어지지 않으면, *getnode()*를 사용하여 하드웨어 주소를 얻습니다. *clock_seq*가 주어지면 시퀀스 번호로 사용합니다. 그렇지 않을 경우, 무작위 14bit 시퀀스 번호를 사용합니다.

uuid.**uuid3**(*namespace*, *name*)
이름 공간 식별자(UUID) 및 이름(문자열)의 MD5 해시를 기반으로 UUID 생성.

uuid.**uuid4**()
무작위 UUID 생성.

uuid.**uuid5**(*namespace*, *name*)
이름 공간 식별자(UUID) 및 이름(문자열)의 SHA-1 해시를 기반으로 UUID 생성.

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**
RFC 4122의 UUID 레이아웃 명시.

uuid.**RESERVED_MICROSOFT**
마이크로소프트 호환성을 위해 예약됨.

uuid.**RESERVED_FUTURE**
추후 정의를 위해 예약됨.

더 보기:

RFC 4122 - 범용 고유 식별자(UUID) URN 이름 공간 이 명세는 UUID를 위한 통합 자원 식별자 이름 공간, UUID의 내부 형식 및 UUID 생성 방법을 정의합니다.

22.20.1 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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> # 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')
```

22.21 socketserver — A framework for network servers

Source code: Lib/socketserver.py

The *socketserver* module simplifies the task of writing network servers.

There are four basic concrete server classes:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the *ForkingMixIn* and *ThreadingMixIn* mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a `with` statement. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket (unless you used a `with` statement).

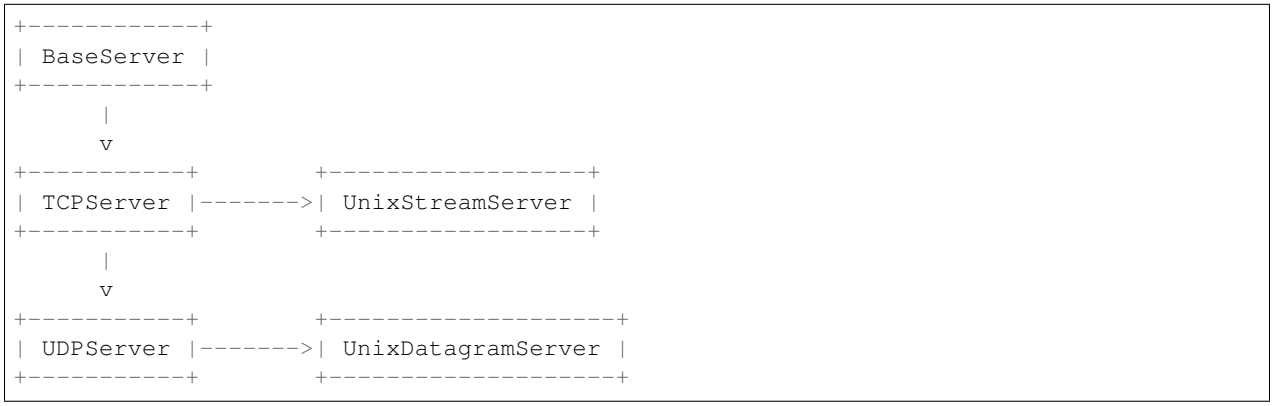
When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which

indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

22.21.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:



Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

class socketserver.**ForkingMixIn**
class socketserver.**ThreadingMixIn**

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

`socketserver.ForkingMixIn.server_close()` waits until all child processes complete, except if `socketserver.ForkingMixIn.block_on_close` attribute is false.

`socketserver.ThreadingMixIn.server_close()` waits until all non-daemon threads complete, except if `socketserver.ThreadingMixIn.block_on_close` attribute is false. Use daemonic threads by setting `ThreadingMixIn.daemon_threads` to `True` to not wait until threads complete.

버전 3.7에서 변경: `socketserver.ForkingMixIn.server_close()` and `socketserver.ThreadingMixIn.server_close()` now waits until all child processes and non-daemonic threads complete. Add a new `socketserver.ForkingMixIn.block_on_close` class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.**ForkingTCPServer**
class socketserver.**ForkingUDPServer**
class socketserver.**ThreadingTCPServer**

class socketserver.**ThreadingUDPServer**
These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *selectors* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See *asyncore* for another way to manage this.

22.21.2 Server Objects

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)
This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()
Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *selectors*, to allow monitoring multiple servers in the same process.

handle_request()
Process a single request. This function calls the following methods in order: *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

serve_forever(*poll_interval=0.5*)
Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

버전 3.3에서 변경: Added `service_actions` call to the `serve_forever` method.

service_actions()
This is called in the *serve_forever()* loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

버전 3.3에 추가.

shutdown()
Tell the *serve_forever()* loop to stop and wait until it does. *shutdown()* must be called while *serve_forever()* is running in a different thread otherwise it will deadlock.

server_close()
Clean up the server. May be overridden.

address_family
The family of protocols to which the server's socket belongs. Common examples are *socket.AF_INET* and *socket.AF_UNIX*.

RequestHandlerClass
The user-provided request handler class; an instance of this class is created for each request.

server_address
The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket
The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address
Whether the server will allow the reuse of an address. This defaults to *False*, and can be set in subclasses to change the policy.

request_queue_size
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to *request_queue_size* requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type
The type of socket used by the server; *socket.SOCK_STREAM* and *socket.SOCK_DGRAM* are two common values.

timeout
Timeout duration, measured in seconds, or *None* if no timeout is desired. If *handle_request()* receives no incoming requests within the timeout period, the *handle_timeout()* method is called.

There are various server methods that can be overridden by subclasses of base server classes like *TCPServer*; these methods aren't useful to external users of the server object.

finish_request(*request*, *client_address*)
Actually processes the request by instantiating *RequestHandlerClass* and calling its *handle()* method.

get_request()
Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(*request*, *client_address*)
This function is called if the *handle()* method of a *RequestHandlerClass* instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

버전 3.6에서 변경: Now only called for exceptions derived from the *Exception* class.

handle_timeout()
This function is called when the *timeout* attribute has been set to a value other than *None* and the timeout

period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)
Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate()
Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()
Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)
Must return a Boolean value; if the value is *True*, the request will be processed, and if it's *False*, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns *True*.

버전 3.6에서 변경: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *server_close()*.

22.21.3 Request Handler Objects

class socketserver.**BaseRequestHandler**
This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()
Called before the *handle()* method to perform any initialization actions required. The default implementation does nothing.

handle()
This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

finish()
Called after the *handle()* method to perform any clean-up actions required. The default implementation does nothing. If *setup()* raises an exception, this function will not be called.

class socketserver.**StreamRequestHandler**
class socketserver.**DatagramRequestHandler**
These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

The `rfile` attributes of both classes support the *io.BufferedIOBase* readable interface, and `DatagramRequestHandler.wfile` supports the *io.BufferedIOBase* writable interface.

버전 3.6에서 변경: `StreamRequestHandler.wfile` also supports the *io.BufferedIOBase* writable interface.

22.21.4 Examples

socketserver.TCPServer Example

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        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 single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side:

```
import socket
import sys
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client:

```
$ 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 Example

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixIn* and *ForkingMixIn* classes.

An example for the *ThreadingMixIn* class:

```
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__":
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        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()
```

The output of the example should look something like this:

```
$ 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
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

22.22 http.server — HTTP servers

Source code: Lib/http/server.py

This module defines classes for implementing HTTP servers (Web servers).

> **경고:** *http.server* is not recommended for production. It only implements *basic security checks*.

One class, *HTTPServer*, is a *socketserver.TCPServer* subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
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*)

This class builds on the *TCPServer* class by storing the server address as instance variables named `server_name` and `server_port`. The server is accessible by the handler, typically through the handler's `server` instance variable.

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)
This class is identical to HTTPServer but uses threads to handle requests by using the *ThreadingMixIn*. This is useful to handle web browsers pre-opening sockets, on which *HTTPServer* would wait indefinitely.

버전 3.7에 추가.

The *HTTPServer* and *ThreadingHTTPServer* must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)
This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). *BaseHTTPRequestHandler* provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the `do_SPAM()` method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

BaseHTTPRequestHandler has the following instance variables:

client_address
Contains a tuple of the form `(host, port)` referring to the client's address.

server
Contains the server instance.

close_connection
Boolean that should be set before *handle_one_request()* returns, indicating if another request may be expected, or if the connection should be shut down.

requestline
Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by *handle_one_request()*. If no valid request line was processed, it should be set to the empty string.

command
Contains the command (request type). For example, `'GET'`.

path
Contains the request path.

request_version
Contains the version string from the request. For example, `'HTTP/1.0'`.

headers
Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request. The `parse_headers()` function from *http.client* is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile
An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

버전 3.6에서 변경: This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler has the following attributes:

server_version
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, `'BaseHTTP/0.2'`.

sys_version
Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, `'Python/1.4'`.

error_message_format
Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

protocol_version
This specifies the HTTP protocol version used in responses. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass
Specifies an *email.message.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `http.client.HTTPMessage`.

responses
This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by *send_response_only()* and *send_error()* methods.

A *BaseHTTPRequestHandler* instance has the following methods:

handle()
Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()
This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()
When a HTTP/1.1 compliant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can chose to send `417 Expectation Failed` as a response header and `return False`.

버전 3.2에 추가.

send_error(*code*, *message=None*, *explain=None*)
Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the *error_message_format* attribute and emitted, after a complete set of headers, as the response body. The *responses* attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

버전 3.4에서 변경: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)
Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the `version_string()` and `date_time_string()` methods, respectively. If the server does not intend to send any other headers using the `send_header()` method, then `send_response()` should be followed by an `end_headers()` call.

버전 3.3에서 변경: Headers are stored to an internal buffer and `end_headers()` needs to be called explicitly.

send_header(*keyword*, *value*)
Adds the HTTP header to an internal buffer which will be written to the output stream when either `end_headers()` or `flush_headers()` is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, `end_headers()` MUST BE called in order to complete the operation.

버전 3.2에서 변경: Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)
Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

버전 3.2에 추가.

end_headers()
Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls `flush_headers()`.

버전 3.2에서 변경: The buffered headers are written to the output stream.

flush_headers()
Finally send the headers to the output stream and flush the internal headers buffer.

버전 3.3에 추가.

log_request(*code='-'*, *size='-'*)
Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(...)
Logs an error when a request cannot be fulfilled. By default, it passes the message to `log_message()`, so it takes the same arguments (*format* and additional values).

log_message(*format*, ...)
Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to `log_message()` are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()
Returns the server software's version string. This is a combination of the `server_version` and `sys_version` attributes.

date_time_string(*timestamp=None*)
Returns the date and time given by *timestamp* (which must be `None` or in the format returned by `time.time()`), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

log_date_time_string()
Returns the current date and time, formatted for logging.

address_string()
Returns the client address.

버전 3.3에서 변경: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)
This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class *BaseHTTPRequestHandler*. This class implements the *do_GET()* and *do_HEAD()* functions.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler*:

server_version
This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.

extensions_map
A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be application/octet-stream. The mapping is used case-insensitively, and so should contain only lower-cased keys.

directory
If not specified, the directory to serve is the current working directory.

The *SimpleHTTPRequestHandler* class defines the following methods:

do_HEAD()
This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the *do_GET()* method for a more complete explanation of the possible headers.

do_GET()
The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named index.html or index.htm (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the list_directory() method. This method uses *os.listdir()* to scan the directory, and returns a 404 error response if the *listdir()* fails.

If the request was mapped to a file, it is opened. Any *OSError* exception in opening the requested file is mapped to a 404, 'File not found' error. If there was a 'If-Modified-Since' header in the request, and the file was not modified after this time, a 304, 'Not Modified' response is sent. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the *extensions_map* variable, and the file contents are returned.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with text/ the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the *test()* function invocation in the *http.server* module.

버전 3.7에서 변경: Support of the 'If-Modified-Since' header.

The *SimpleHTTPRequestHandler* class can be used in the following manner in order to create a very basic webserver serving files relative to the current directory:

```
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()
```

http.server can also be invoked directly using the `-m` switch of the interpreter with a `port number` argument. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server 8000
```

By default, server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. For example, the following command causes the server to bind to localhost only:

```
python -m http.server 8000 --bind 127.0.0.1
```

버전 3.4에 추가: `--bind` argument was introduced.

By default, server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

버전 3.7에 추가: `--directory` specify alternate directory

class `http.server.`**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in *SimpleHTTPRequestHandler*.

참고: CGI scripts run by the *CGIHTTPRequestHandler* class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The *CGIHTTPRequestHandler* defines the following data member:

cgi_directories

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The *CGIHTTPRequestHandler* defines the following method:

do_POST()

This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

CGIHTTPRequestHandler can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi 8000
```

22.22.1 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 3.7.16에 추가: scrubbing control characters from log messages

22.23 `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 않으며 쿠키 처리와 관련하여 오늘날의 많은 브라우저와 서버가 구문 분석 규칙을 완화했다는 사실이 발견되었습니다. 결과적으로, 사용되는 구문 분석 규칙은 약간 덜 엄격합니다.

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.

22.23.1 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
```

22.23.2 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

httponly 어트리뷰트는 쿠키가 HTTP 요청으로만 전송되고 자바스크립트를 통해 액세스할 수 없도록 지정합니다. 이것은 교차 사이트 스크립팅의 일부 형식을 방지하기 위한 것입니다.

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.

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 닥셔너리 *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.

22.23.3 예제

다음 예제는 *http.cookies* 모듈을 사용하는 방법을 보여줍니다.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
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") # load from a string (HTTP header)
>>> 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)
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 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

22.24 http.cookiejar — Cookie handling for HTTP clients

Source code: Lib/http/cookiejar.py

The *http.cookiejar* module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – *cookies* – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by **RFC 2965** are handled. RFC 2965 handling is switched off by default. **RFC 2109** cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the Internet are Netscape cookies. *http.cookiejar* attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

참고: The various named parameters found in *Set-Cookie* and *Set-Cookie2* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception:

exception http.cookiejar.**LoadError**
Instances of *FileCookieJar* raise this exception on failure to load cookies from a file. *LoadError* is a subclass of *OSError*.

버전 3.3에서 변경: LoadError was made a subclass of *OSError* instead of *IOError*.

The following classes are provided:

class http.cookiejar.**CookieJar**(*policy=None*)
policy is an object implementing the *CookiePolicy* interface.

The *CookieJar* class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. *CookieJar* instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class http.cookiejar.**FileCookieJar**(*filename*, *delayload=None*, *policy=None*)
policy is an object implementing the *CookiePolicy* interface. For the other arguments, see the documentation for the corresponding attributes.

A *CookieJar* which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the *load()* or *revert()* method is called. Subclasses of this class are documented in section *FileCookieJar subclasses and co-operation with web browsers*.

class http.cookiejar.**CookiePolicy**
This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

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*)
Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain

names that we never accept cookies from, nor return cookies to. *allowed_domains* if not *None*, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for *CookiePolicy* and *DefaultCookiePolicy* objects.

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or *rfc2109_as_netscape* is `True`, RFC 2109 cookies are 'downgraded' by the *CookieJar* instance to Netscape cookies, by setting the `version` attribute of the *Cookie* instance to 0. *DefaultCookiePolicy* also provides some parameters to allow some fine-tuning of policy.

class http.cookiejar.**Cookie**
This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of *http.cookiejar* construct their own *Cookie* instances. Instead, if necessary, call `make_cookies()` on a *CookieJar* instance.

더 보기:

Module *urllib.request* URL opening with automatic cookie handling.

Module *http.cookies* HTTP cookie classes, principally useful for server-side code. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and *http.cookiejar*) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism Obsoleted by **RFC 2965**. Uses *Set-Cookie* with version=1.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses *Set-Cookie2* in place of *Set-Cookie*. Not widely used.

http://kristol.org/cookie/errata.html Unfinished errata to **RFC 2965**.

RFC 2964 - Use of HTTP State Management

22.24.1 CookieJar and FileCookieJar Objects

CookieJar objects support the *iterator* protocol for iterating over contained *Cookie* objects.

CookieJar has the following methods:

CookieJar.**add_cookie_header**(*request*)
Add correct *Cookie* header to *request*.

If policy allows (ie. the `rfc2965` and `hide_cookie2` attributes of the *CookieJar*'s *CookiePolicy* instance are true and false respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `get_host()`, `get_type()`, `unverifiable()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and `origin_req_host` attribute as documented by *urllib.request*.

버전 3.3에서 변경: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

CookieJar.**extract_cookies**(*response*, *request*)
Extract cookies from HTTP *response* and store them in the *CookieJar*, where allowed by policy.

The *CookieJar* will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the *CookiePolicy.set_ok()* method's approval).

The *response* object (usually the result of a call to *urllib.request.urlopen()*, or similar) should support an `info()` method, which returns an *email.message.Message* instance.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `origin_req_host` attribute, as documented by *urllib.request*. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

버전 3.3에서 변경: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.`**`set_policy`**(*policy*)
Set the *CookiePolicy* instance to be used.

`CookieJar.`**`make_cookies`**(*response*, *request*)
Return sequence of *Cookie* objects extracted from *response* object.

See the documentation for *extract_cookies()* for the interfaces required of the *response* and *request* arguments.

`CookieJar.`**`set_cookie_if_ok`**(*cookie*, *request*)
Set a *Cookie* if policy says it's OK to do so.

`CookieJar.`**`set_cookie`**(*cookie*)
Set a *Cookie*, without checking with policy to see whether or not it should be set.

`CookieJar.`**`clear`**([*domain*[, *path*[, *name*]]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises *KeyError* if no matching cookie exists.

`CookieJar.`**`clear_session_cookies`**()
Discard all session cookies.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

FileCookieJar implements the following additional methods:

`FileCookieJar.`**`save`**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Save cookies to a file.

This base class raises *NotImplementedError*. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is *None*, *ValueError* is raised.

ignore_discard: save even cookies set to be discarded. *ignore_expires*: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the *load()* or *revert()* methods.

`FileCookieJar.`**`load`**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for *save()*.

The named file must be in the format understood by the class, or *LoadError* will be raised. Also, *OSError* may be raised, for example if the file does not exist.

버전 3.3에서 변경: *IOError* used to be raised, it is now an alias of *OSError*.

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as *load()*. If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar.**filename**
Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.**delayload**
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A *CookieJar* object may ignore it. None of the *FileCookieJar* classes included in the standard library lazily loads cookies.

22.24.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

class http.cookiejar.**MozillaCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by the Lynx and Netscape browsers).

참고: This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

> 경고: Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class http.cookiejar.**LWPCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

22.24.3 CookiePolicy Objects

Objects implementing the *CookiePolicy* interface have the following methods:

CookiePolicy.**set_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.extract_cookies()*.

CookiePolicy.**return_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be returned to server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.add_cookie_header()*.

CookiePolicy.**domain_return_ok**(*domain*, *request*)
Return `False` if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from *domain_return_ok()* and *path_return_ok()* leaves all the work to *return_ok()*.

If *domain_return_ok()* returns true for the cookie domain, *path_return_ok()* is called for the cookie path. Otherwise, *path_return_ok()* and *return_ok()* are never called for that cookie domain. If *path_return_ok()* returns true, *return_ok()* is called with the *Cookie* object itself for a full check. Otherwise, *return_ok()* is never called for that cookie path.

Note that *domain_return_ok()* is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for *path_return_ok()*.

The *request* argument is as documented for *return_ok()*.

CookiePolicy.**path_return_ok**(*path*, *request*)
Return `False` if cookies should not be returned, given cookie path.

See the documentation for *domain_return_ok()*.

In addition to implementing the methods above, implementations of the *CookiePolicy* interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.**netscape**
Implement Netscape protocol.

CookiePolicy.**rfc2965**
Implement **RFC 2965** protocol.

CookiePolicy.**hide_cookie2**
Don't add *Cookie2* header to requests (the presence of this header indicates to the server that we understand **RFC 2965** cookies).

The most useful way to define a *CookiePolicy* class is by subclassing from *DefaultCookiePolicy* and overriding some or all of the methods above. *CookiePolicy* itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

22.24.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the *CookiePolicy* interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for *allowed_domains*). If you set a whitelist, you can turn it off again by setting it to *None*.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blacklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains `"192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

`DefaultCookiePolicy.`**`blocked_domains`**`()`
Return the sequence of blocked domains (as a tuple).

`DefaultCookiePolicy.`**`set_blocked_domains`**`(`*blocked_domains*`)`
Set the sequence of blocked domains.

`DefaultCookiePolicy.`**`is_blocked`**`(`*domain*`)`
Return whether *domain* is on the blacklist for setting or receiving cookies.

`DefaultCookiePolicy.`**`allowed_domains`**`()`
Return *None*, or the sequence of allowed domains (as a tuple).

`DefaultCookiePolicy.`**`set_allowed_domains`**`(`*allowed_domains*`)`
Set the sequence of allowed domains, or *None*.

`DefaultCookiePolicy.`**`is_not_allowed`**`(`*domain*`)`
Return whether *domain* is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

`DefaultCookiePolicy.`**`rfc2109_as_netscape`**
If true, request that the *CookieJar* instance downgrade **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the *Cookie* instance to 0. The default value is *None*, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

`DefaultCookiePolicy.`**`strict_domain`**
Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

`DefaultCookiePolicy.`**`strict_rfc2965_unverifiable`**
Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a

redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

`DefaultCookiePolicy.`**`strict_ns_unverifiable`**
Apply **RFC 2965** rules on unverifiable transactions even to Netscape cookies.

`DefaultCookiePolicy.`**`strict_ns_domain`**
Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

`DefaultCookiePolicy.`**`strict_ns_set_initial_dollar`**
Ignore cookies in Set-Cookie: headers that have names starting with `'$'`.

`DefaultCookiePolicy.`**`strict_ns_set_path`**
Don't allow setting cookies whose path doesn't path-match request URI.

`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`**
When setting cookies, the 'host prefix' must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

`DefaultCookiePolicy.`**`DomainStrictNonDomain`**
Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no `domain` cookie-attribute).

`DefaultCookiePolicy.`**`DomainRFC2965Match`**
When setting cookies, require a full **RFC 2965** domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

`DefaultCookiePolicy.`**`DomainLiberal`**
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

`DefaultCookiePolicy.`**`DomainStrict`**
Equivalent to `DomainStrictNoDots|DomainStrictNonDomain`.

22.24.5 Cookie Objects

`Cookie` instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because **RFC 2109** cookies may be 'downgraded' by *`http.cookiejar`* from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *`CookiePolicy`* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

`Cookie.`**`version`**
Integer or *`None`*. Netscape cookies have *`version`* 0. **RFC 2965** and **RFC 2109** cookies have a `version` cookie-attribute of 1. However, note that *`http.cookiejar`* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *`version`* is 0.

`Cookie.`**`name`**
Cookie name (a string).

`Cookie.`**`value`**
Cookie value (a string), or *`None`*.

Cookie.**port**
String representing a port or a set of ports (eg. '80', or '80,8080'), or *None*.

Cookie.**path**
Cookie path (a string, eg. `'/acme/rocket_launchers'`).

Cookie.**secure**
`True` if cookie should only be returned over a secure connection.

Cookie.**expires**
Integer expiry date in seconds since epoch, or *None*. See also the *is_expired()* method.

Cookie.**discard**
`True` if this is a session cookie.

Cookie.**comment**
String comment from the server explaining the function of this cookie, or *None*.

Cookie.**comment_url**
URL linking to a comment from the server explaining the function of this cookie, or *None*.

Cookie.**rfc2109**
`True` if this cookie was received as an **RFC 2109** cookie (ie. the cookie arrived in a *Set-Cookie* header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because *http.cookiejar* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

Cookie.**port_specified**
`True` if a port or set of ports was explicitly specified by the server (in the *Set-Cookie* / *Set-Cookie2* header).

Cookie.**domain_specified**
`True` if a domain was explicitly specified by the server.

Cookie.**domain_initial_dot**
`True` if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.**has_nonstandard_attr**(*name*)
Return `True` if cookie has the named cookie-attribute.

Cookie.**get_nonstandard_attr**(*name*, *default=None*)
If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

Cookie.**set_nonstandard_attr**(*name*, *value*)
Set the value of the named cookie-attribute.

The *Cookie* class also defines the following method:

Cookie.**is_expired**(*now=None*)
`True` if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

22.24.6 Examples

The first example shows the most common usage of *http.cookiejar*:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
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/")
```

The next example illustrates the use of *DefaultCookiePolicy*. Turn on **RFC 2965** cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
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/")
```

22.25 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*

22.26 xmlrpc.client — XML-RPC client access

Source code: Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

경고: The *xmlrpc.client* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 취약점*.

버전 3.5에서 변경: For HTTPS URIs, *xmlrpc.client* now performs all the necessary certificate and hostname checks by default.

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, *, *context=None*)

버전 3.3에서 변경: The *use_builtin_types* flag was added.

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be *ssl.SSLContext* and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
`boolean`	*bool*
`int`, `i1`, `i2`, `i4`, `i8` or `biginteger`	*int* in range from -2147483648 to 2147483647. Values get the `<int>` tag.
`double` or `float`	*float*. Values get the `<double>` tag.
`string`	*str*
`array`	*list* or *tuple* containing conformable elements. Arrays are returned as *lists*.
`struct`	*dict*. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their *__dict__* attribute is transmitted.
`dateTime.iso8601`	*DateTime* or *datetime.datetime*. Returned type depends on values of *use_builtin_types* and *use_datetime* flags.
`base64`	*Binary*, *bytes* or *bytearray*. Returned type depends on the value of the *use_builtin_types* flag.
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.
`bigdecimal`	*decimal.Decimal*. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special *Fault* instance, used to signal XML-RPC server errors, or *ProtocolError* used to signal an error in the HTTP/HTTPS transport layer. Both *Fault* and *ProtocolError* derive from a base class called `Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use *bytes* or *bytearray* classes or the *Binary* wrapper class described below.

`Server` is retained as an alias for *ServerProxy* for backwards compatibility. New code should use *ServerProxy*.

버전 3.5에서 변경: Added the *context* argument.

버전 3.6에서 변경: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See http://ws.apache.org/xmlrpc/types.html for a description.

더 보기:

XML-RPC HOWTO A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification The official specification.

Unofficial XML-RPC Errata Fredrik Lundh's "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

22.26.1 ServerProxy Objects

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.`**`listMethods`**`()`
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.`**`methodSignature`**`(`*name*`)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

`ServerProxy.system.`**`methodHelp`**`(`*name*`)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

버전 3.5에서 변경: Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code:

```
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()
```

The client code for the preceding server:

```
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)))
```

22.26.2 DateTime Objects

class xmlrpc.client.**DateTime**

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through rich comparison and `__repr__()` methods.

A working example follows. The server code:

```
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()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

22.26.3 Binary Objects

class xmlrpc.client.**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute:

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
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()
```

The client gets the image and saves it to a file:

```
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)
```

22.26.4 Fault Objects

class xmlrpc.client.**Fault**

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

A string indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
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()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

22.26.5 ProtocolError Objects

class xmlrpc.client.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url
The URI or URL that triggered the error.

errcode
The error code.

errmsg
The error message or diagnostic string.

headers
A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
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)
```

22.26.6 MultiCall Objects

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpc.client.**MultiCall**(*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

[1] This approach has been first presented in a discussion on xmlrpc.com.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
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()
```

The client code for the preceding server:

```
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))
```

22.26.7 Convenience Functions

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

버전 3.3에서 변경: The *use_builtin_types* flag was added.

22.26.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
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))
```

22.26.9 Example of Client and Server Usage

See *SimpleXMLRPCServer 예제*.

22.27 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: *xmlrpc.server* 모듈은 악의적으로 구성된 데이터로부터 안전하지 않습니다. 신뢰할 수 없거나 인증되지 않은 데이터를 구문 분석해야 한다면 *XML 취약점*를 참조하십시오.

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.

22.27.1 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

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    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))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
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),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @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:
        @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
```

22.27.2 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()
```

22.27.3 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.

22.27.4 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.

22.27.5 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.

22.28 ipaddress — IPv4/IPv6 manipulation library

Source code: Lib/ipaddress.py

---

ipaddress provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see ipaddress-howto.

버전 3.3에 추가.

22.28.1 Convenience factory functions

The *ipaddress* module provides factory functions to conveniently create IP addresses, networks and interfaces:

ipaddress.**ip_address**(*address*)
Return an *IPv4Address* or *IPv6Address* object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> 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*)
Return an *IPv4Network* or *IPv6Network* object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than 2**32 will be considered to be IPv4 by default. *strict* is passed to *IPv4Network* or *IPv6Network* constructor. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)
Return an *IPv4Interface* or *IPv6Interface* object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don't know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

22.28.2 IP Addresses

Address objects

The *IPv4Address* and *IPv6Address* objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by *IPv4Address* objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Address**(*address*)
Construct an IPv4 address. An *AddressValueError* is raised if *address* is not a valid IPv4 address.

The following constitutes a valid IPv4 address:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0–255, separated by dots (e.g. `192.168.0.1`). Each integer represents an octet (byte) in the address. Leading zeroes are tolerated only for values less than 8 (as there is no ambiguity between the decimal and octal interpretations of such strings).
2. An integer that fits into 32 bits.
3. An integer packed into a *bytes* object of length 4 (most significant octet first).

```
>>> 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')
```

version
The appropriate version number: `4` for IPv4, `6` for IPv6.

max_prefixlen
The total number of bits in the address representation for this version: `32` for IPv4, `128` for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

compressed

exploded
The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed
The binary representation of this address - a *bytes* object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

reverse_pointer
The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> 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'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

버전 3.5에 추가.

is_multicast
`True` if the address is reserved for multicast use. See **RFC 3171** (for IPv4) or **RFC 2373** (for IPv6).

is_private
`True` if the address is allocated for private networks. See iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6).

is_global
`True` if the address is allocated for public networks. See iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6).

버전 3.4에 추가.

is_unspecified
`True` if the address is unspecified. See **RFC 5735** (for IPv4) or **RFC 2373** (for IPv6).

is_reserved
`True` if the address is otherwise IETF reserved.

is_loopback
`True` if this is a loopback address. See **RFC 3330** (for IPv4) or **RFC 2373** (for IPv6).

is_link_local
`True` if the address is reserved for link-local usage. See **RFC 3927**.

class ipaddress.**IPv6Address**(*address*)
Construct an IPv6 address. An *AddressValueError* is raised if *address* is not a valid IPv6 address.

The following constitutes a valid IPv6 address:

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See **RFC 4291** for details. For example, `"0000:0000:0000:0000:0000:0abc:0007:0def"` can be compressed to `"::abc:7:def"`.
2. An integer that fits into 128 bits.
3. An integer packed into a *bytes* object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by `str(addr)` for IPv6 addresses.

exploded

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes, see the corresponding documentation of the *IPv4Address* class:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

is_unspecified

is_reserved

is_loopback

is_link_local
버전 3.4에 추가: is_global

is_site_local
`True` if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by **RFC 3879**. Use *is_private* to test if this address is in the space of unique local addresses as defined by **RFC 4193**.

ipv4_mapped
For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

sixtofour
For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by **RFC 3056**, this property will report the embedded IPv4 address. For any other address, this property will be `None`.

teredo
For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by **RFC 4380**, this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the *str()* and *int()* builtin functions:

```
>>> 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
```

Operators

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Some examples:

```
>>> 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
```

Arithmetic operators

Integers can be added to or subtracted from address objects. Some examples:

```
>>> 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
```

22.28.3 IP Network definitions

The *IPv4Network* and *IPv6Network* objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between *IPv4Network* and *IPv6Network*, so to avoid duplication they are only documented for *IPv4Network*. Network objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.

3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.

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`).

An *AddressValueError* is raised if *address* is not a valid IPv4 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise *TypeError* if the argument's IP version is incompatible to `self`.

버전 3.5에서 변경: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen
Refer to the corresponding attribute documentation in *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local
These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address
The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address
The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask
The host mask, as an *IPv4Address* object.

netmask
The net mask, as an *IPv4Address* object.

with_prefixlen

compressed

exploded
A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask
A string representation of the network, with the mask in net mask notation.

with_hostmask
A string representation of the network, with the mask in host mask notation.

num_addresses
The total number of addresses in the network.

prefixlen
Length of the network prefix, in bits.

hosts()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result.

```
>>> 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')]
```

overlaps(*other*)
`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)
Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises `ValueError` if *network* is not completely contained in this network.

```
>>> 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*)
The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> 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'),
 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*)
The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> 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*)
Return `True` if this network is a subnet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

버전 3.7에 추가.

supernet_of(*other*)
Return `True` if this network is a supernet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

버전 3.7에 추가.

compare_networks(*other*)
Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren't. Returns either `-1`, `0` or `1`.

```
>>> 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 폐지: It uses the same ordering and comparison algorithm as "<", "==", and ">"

class ipaddress.**IPv6Network**(*address*, *strict=True*)
Construct an IPv6 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.

3. An integer packed into a *bytes* object of length 16, big-endian. The interpretation is similar to an integer *address*.

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.

An *AddressValueError* is raised if *address* is not a valid IPv6 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

버전 3.5에서 변경: Added the two-tuple form for the *address* constructor parameter.

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()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result.

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*)
Refer to the corresponding attribute documentation in *IPv4Network*.

is_site_local
These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

Operators

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Iteration

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the *hosts()* method). An example:

```
>>> 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')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```
>>> 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
```

22.28.4 Interface objects

Interface objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Interface**(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of *IPv4Network*, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of *IPv4Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (*IPv4Address*) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask
Refer to the corresponding attribute documentation in *IPv4Interface*.

Operators

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (== and !=), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (<, >, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

22.28.5 Other Module Level Functions

The module also provides the following module level functions:

ipaddress.**v4_int_to_packed**(*address*)
Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> 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*)
Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.**summarize_address_range**(*first*, *last*)
Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first *IPv4Address* or *IPv6Address* in the range and *last* is the last *IPv4Address* or *IPv6Address* in the range. A *TypeError* is raised if *first* or *last* are not IP addresses or are not of the same version. A *ValueError* is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [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*)
Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an iterator of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [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*)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the *key* argument to *sorted()*.

obj is either a network or address object.

22.28.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

exception ipaddress.**AddressValueError**(*ValueError*)

Any value error related to the address.

exception ipaddress.**NetmaskValueError**(*ValueError*)

Any value error related to the net mask.

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 `audioop` — Manipulate raw audio data

The `audioop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16, 24 or 32 bits wide, stored in *bytes-like objects*. All scalar items are integers, unless specified otherwise.

버전 3.4에서 변경: Support for 24-bit samples was added. All functions now accept any *bytes-like object*. String input now results in an immediate error.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

exception audioop.**error**
This exception is raised on all errors, such as unknown number of bytes per sample, etc.

audioop.**add**(*fragment1*, *fragment2*, *width*)
Return a fragment which is the addition of the two samples passed as parameters. *width* is the sample width in bytes, either `1`, `2`, `3` or `4`. Both fragments should have the same length. Samples are truncated in case of overflow.

audioop.**adpcm2lin**(*adpcmfragment*, *width*, *state*)
Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm()` for details on ADPCM coding. Return a tuple `(sample, newstate)` where the sample has the width specified in *width*.

audioop.**alaw2lin**(*fragment*, *width*)
Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

audioop.**avg**(*fragment*, *width*)
Return the average over all samples in the fragment.

audioop.**avgpp**(*fragment*, *width*)
Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

audioop.**bias**(*fragment*, *width*, *bias*)
Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

audioop.**byteswap**(*fragment*, *width*)
"Byteswap" all samples in a fragment and returns the modified fragment. Converts big-endian samples to little-endian and vice versa.

버전 3.4에 추가.

audioop.**cross**(*fragment*, *width*)
Return the number of zero crossings in the fragment passed as an argument.

audioop.**findfactor**(*fragment*, *reference*)
Return a factor *F* such that `rms(add(fragment, mul(reference, -F)))` is minimal, i.e., return the factor with which you should multiply *reference* to make it match as well as possible to *fragment*. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to `len(fragment)`.

audioop.**findfit**(*fragment*, *reference*)
Try to match *reference* as well as possible to a portion of *fragment* (which should be the longer fragment). This is (conceptually) done by taking slices out of *fragment*, using *findfactor()* to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple `(offset, factor)` where *offset* is the (integer) offset into *fragment* where the optimal match started and *factor* is the (floating-point) factor as per *findfactor()*.

audioop.**findmax**(*fragment*, *length*)
Search *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e., return *i* for which `rms(fragment[i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to `len(fragment)`.

audioop.**getsample**(*fragment*, *width*, *index*)
Return the value of sample *index* from the fragment.

audioop.**lin2adpcm**(*fragment*, *width*, *state*)
Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple `(adpcmfrag, newstate)`, and the *newstate* should be passed to the next call of *lin2adpcm()*. In the initial call, `None` can be passed as the state. *adpcmfrag* is the ADPCM coded fragment packed 2 4-bit values per byte.

audioop.**lin2alaw**(*fragment*, *width*)
Convert samples in the audio fragment to a-LAW encoding and return this as a bytes object. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)
Convert samples between 1-, 2-, 3- and 4-byte formats.

참고: In some audio formats, such as .WAV files, 16, 24 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

The same, in reverse, has to be applied when converting from 8 to 16, 24 or 32 bit width samples.

---

audioop.**lin2ulaw**(*fragment*, *width*)
Convert samples in the audio fragment to u-LAW encoding and return this as a bytes object. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.**max**(*fragment*, *width*)
Return the maximum of the *absolute value* of all samples in a fragment.

audioop.**maxpp**(*fragment*, *width*)
Return the maximum peak-peak value in the sound fragment.

audioop.**minmax**(*fragment*, *width*)
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

audioop.**mul**(*fragment*, *width*, *factor*)
Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Samples are truncated in case of overflow.

audioop.**ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])
Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple `(newfragment, newstate)`, and *newstate* should be passed to the next call of *ratecv()*. The initial call should pass `None` as the state.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to `1` and `0` respectively.

audioop.**reverse**(*fragment*, *width*)
Reverse the samples in a fragment and returns the modified fragment.

audioop.**rms**(*fragment*, *width*)
Return the root-mean-square of the fragment, i.e. `sqrt(sum(S_i^2)/n)`.

This is a measure of the power in an audio signal.

audioop.**tomono**(*fragment*, *width*, *lfactor*, *rfactor*)
Convert a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *rfactor* before adding the two channels to give a mono signal.

audioop.**tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)
Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

audioop.**ulaw2lin**(*fragment*, *width*)
Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as *mul()* or *max()* make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the *initial* state (the one you passed to *lin2adpcm()*) along to the decoder, not the final state (as returned by the coder). If you want to use *struct.Struct* to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The `find*()` routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

23.2 `aifc` — AIFF와 AIFC 파일 읽고 쓰기

소스 코드: Lib/aifc.py

이 모듈은 AIFF와 AIFF-C 파일을 읽고 쓸 수 있도록 지원합니다. AIFF는 디지털 오디오 샘플을 파일에 저장하는 형식인 Audio Interchange File Format입니다. AIFF-C는 오디오 데이터를 압축하는 기능을 포함하는 이 형식의 새 버전입니다.

오디오 파일에는 오디오 데이터를 설명하는 많은 파라미터가 있습니다. 샘플링 속도나 프레임 속도는 음향을 샘플링하는 초당 횟수입니다. 채널 수는 오디오가 모노(mono), 스테레오(stereo) 또는 쿼드로(quadro) 인지를 나타냅니다. 각 프레임은 채널 당 하나의 샘플로 구성됩니다. 샘플 크기는 각 샘플의 크기를 바이트로 표현한 것입니다. 따라서 프레임은 `nchannels * samplesize` 바이트로 구성되고, 1초 분량의 오디오는 `nchannels * samplesize * framerate` 바이트로 구성됩니다.

예를 들어, CD 품질 오디오는 샘플 크기가 2바이트 (16비트)이고, 2채널(스테레오)을 사용하며 프레임 속도가 44,100프레임/초입니다. 이는 4바이트(2*2)의 프레임 크기를 제공하고, 1초 분량의 오디오는 2*2*44100바이트 (176,400바이트)를 차지합니다.

모듈 *aifc*는 다음 함수를 정의합니다:

aifc.**open**(*file*, *mode=None*)
AIFF나 AIFF-C 파일을 열고 아래에 설명된 메서드를 갖는 객체 인스턴스를 반환합니다. 인자 *file*는 파일을 명명하는 문자열이거나 파일 객체입니다. *mode*는 파일을 읽기 위해 열어야 할 때 `'r'`이나 `'rb'` 여야 하며, 파일을 쓰기 위해 열어야 하는 경우 `'w'`나 `'wb'` 여야 합니다. 생략하면, `file.mode`가 있으면 그것을 쓰고, 그렇지 않으면 `'rb'`가 사용됩니다. 쓰기 위해 열 때는, 앞으로 기록할 샘플의 총수를 미리 알고 `writeframesraw()` 및 `setnframes()`를 사용하지 않는 한 파일 객체는 위치 변경할 수 있어야 (seekable) 합니다. *open()* 함수는 `with` 문에서 사용될 수 있습니다. `with` 블록이 완료될 때 *close()* 메서드가 호출됩니다.

버전 3.4에서 변경: `with` 문에 대한 지원이 추가되었습니다.

파일을 읽기 위해 열 때 *open()*가 반환하는 객체는 다음과 같은 메서드를 가집니다:

aifc.**getnchannels**()
오디오 채널 수를 반환합니다 (모노는 1, 스테레오는 2).

aifc.**getsampwidth**()
개별 샘플의 크기를 바이트 단위로 반환합니다.

aifc.**getframerate**()
샘플링 속도(초당 오디오 프레임의 수)를 반환합니다.

aifc.**getnframes**()
파일 내의 오디오 프레임의 수를 반환합니다.

aifc.**getcomptype**()
오디오 파일에서 사용된 압축 유형을 설명하는 길이 4의 바이트열을 반환합니다. AIFF 파일의 경우, 반환 값은 `b'NONE'`입니다.

aifc.**getcompname**()
오디오 파일에서 사용된 압축 유형을 사람이 읽을 수 있는 형식으로 변환할 수 있는 바이트열을 반환합니다. AIFF 파일의 경우, 반환 값은 `b'not compressed'`입니다.

aifc.**getparams**()
`get*()` 메서드의 결과와 동등한, *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`을 반환합니다.

aifc.**getmarkers**()
오디오 파일의 마커(marker) 리스트를 반환합니다. 마커는 세 요소의 튜플로 구성됩니다. 첫 번째는 마크 ID(정수)이고, 두 번째는 데이터 시작부터 따진 프레임에서의 마크 위치이며 (정수), 세 번째는 마크의 이름입니다 (문자열).

aifc.**getmark**(*id*)
지정된 *id*를 가진 마크에 대해 *getmarkers()*에 설명된 튜플을 반환합니다.

aifc.**readframes**(*nframes*)
오디오 파일에서 다음 *nframes* 프레임을 읽고 반환합니다. 반환된 데이터는 각 프레임의 모든 채널의 압축되지 않은 샘플을 포함하는 문자열입니다.

aifc.**rewind**()
읽기 포인터를 되감습니다. 다음 *readframes()*는 처음부터 시작됩니다.

aifc.**setpos**(*pos*)
지정된 프레임 번호로 위치 변경합니다.

aifc.**tell**()
현재의 프레임 번호를 반환합니다.

aifc.**close**()
AIFF 파일을 닫습니다. 이 메서드를 호출한 후에는 객체를 더는 사용할 수 없습니다.

파일을 쓰기 위해 열 때, *open()*이 반환하는 객체는 readframes()와 setpos()를 제외하고 위의 모든 메서드를 가집니다. 이에 더해 다음과 같은 메서드가 있습니다. get*() 메서드는 해당 set*() 메서드가 호출된 후에만 호출할 수 있습니다. 첫 번째 writeframes()나 writeframesraw() 이전에, 프레임 수를 제외한 모든 파라미터를 채워야 합니다.

aifc.**aiff**()
AIFF 파일을 만듭니다. 기본값은 AIFF-C 파일을 만드는 것입니다. 파일 이름이 '.aiff'로 끝날 때는 예외인데, 이때 기본값은 AIFF 파일입니다.

aifc.**aifc**()
AIFF-C 파일을 만듭니다. 기본값은 AIFF-C 파일을 만드는 것입니다. 파일 이름이 '.aiff'로 끝날 때는 예외인데, 이때 기본값은 AIFF 파일입니다.

aifc.**setnchannels**(*nchannels*)
오디오 파일의 채널 수를 지정합니다.

aifc.**setsampwidth**(*width*)
오디오 샘플의 크기를 바이트 단위로 지정합니다.

aifc.**setframerate**(*rate*)
샘플링 빈도를 초당 프레임 수로 지정합니다.

aifc.**setnframes**(*nframes*)
오디오 파일에 기록할 프레임 수를 지정합니다. 이 파라미터가 설정되지 않거나, 올바르게 설정되지 않으면, 파일은 위치 변경을 지원해야 합니다.

aifc.**setcomptype**(*type*, *name*)
압축 유형을 지정합니다. 지정하지 않으면, 오디오 데이터는 압축되지 않습니다. AIFF 파일에서, 압축은 불가능합니다. name 매개 변수는 사람이 읽을 수 있는 압축 유형 설명을 바이트열로 제공해야 하며, type 매개 변수는 길이가 4인 바이트열이어야 합니다. 현재 다음 압축 유형이 지원됩니다: b'NONE', b'ULAW',b'ALAW',b'G722'.

aifc.**setparams**(*nchannels*, *sampwidth*, *framerate*, *comptype*, *compname*)
위의 모든 파라미터를 한 번에 설정합니다. 인자는 여러 파라미터로 구성된 튜플입니다. 이것은 *getparams()* 호출의 결과를 *setparams()*의 인자로 사용할 수 있음을 뜻합니다.

aifc.**setmark**(*id*, *pos*, *name*)
지정된 id(0보다 큰 값)의 마크를 주어진 name으로 주어진 위치에 추가합니다. 이 메서드는 *close()* 이전에 언제든지 호출할 수 있습니다.

aifc.**tell**()
출력 파일의 현재 쓰기 위치를 반환합니다. *setmark()*와 함께 사용하면 유용합니다.

aifc.**writeframes**(*data*)
데이터를 출력 파일에 씁니다. 이 메서드는 오디오 파일 파라미터가 설정된 후에만 호출할 수 있습니다.

버전 3.4에서 변경: 이제 모든 바이트열류 객체가 허용됩니다.

aifc.**writeframesraw**(*data*)
오디오 파일의 헤더가 갱신되지 않는다는 점을 제외하고는, *writeframes()*와 같습니다.

버전 3.4에서 변경: 이제 모든 바이트열류 객체가 허용됩니다.

aifc.**close**()
AIFF 파일을 닫습니다. 파일의 헤더는 오디오 데이터의 실제 크기를 반영하도록 갱신됩니다. 이 메서드를 호출한 후에는, 객체를 더는 사용할 수 없습니다.

23.3 sunau — Sun AU 파일 읽고 쓰기

소스 코드: Lib/sunau.py

---

sunau 모듈은 Sun AU 음향 형식에 편리한 인터페이스를 제공합니다. 이 모듈은 모듈 *aifc*와 *wave* 모듈과 인터페이스 호환됩니다.

오디오 파일은 헤더와 뒤따르는 데이터로 구성됩니다. 헤더의 필드는 다음과 같습니다:

필드	내용
매직 워드	4바이트 `.snd`.
헤더 크기	info를 포함한 헤더의 크기 (바이트).
데이터 크기	데이터의 물리적 크기 (바이트).
인코딩(encoding)	오디오 샘플이 인코딩되는 방법을 나타냅니다.
샘플 속도(sample rate)	샘플링 속도.
채널 수	샘플의 채널 수.
info(정보)	오디오 파일에 대한 설명을 제공하는 ASCII 문자열 (널 바이트로 채워집니다).

info 필드는 제외하고, 모든 헤더 필드의 크기는 4바이트입니다. 이것들은 모두 빅 엔디안 바이트 순서로 인코딩된 32비트 부호 없는 정수입니다.

sunau 모듈은 다음 함수를 정의합니다:

sunau.**open**(*file*, *mode*)
: *file*이 문자열이면, 그 이름으로 파일을 열고, 그렇지 않으면 위치 변경할 수 있는 파일류 객체로 처리합니다. *mode*는 다음 중 하나일 수 있습니다.

  'r' 읽기 전용 모드.

  'w' 쓰기 전용 모드.

  읽기와 쓰기를 동시에 지원하지 않음에 유의하십시오.

  `'r'`의 *mode*는 `AU_read` 객체를 반환하고, `'w'` 나 `'wb'`의 *mode*는 `AU_write` 객체를 반환합니다.

sunau.**openfp**(*file*, *mode*)
: 이전 버전과의 호환성을 위해 유지되는, *open()*의 동의어입니다.

  Deprecated since version 3.7, will be removed in version 3.9.

sunau 모듈은 다음 예외를 정의합니다:

exception sunau.**Error**
: Sun AU 명세나 구현 결함으로 인해 무언가가 불가능할 때 발생하는 에러.

sunau 모듈은 다음 데이터 항목을 정의합니다:

sunau.**AUDIO_FILE_MAGIC**
: 유효한 모든 Sun AU 파일이 시작하는 빅 엔디안 형식으로 저장된 정수. 이것은 정수로 해석되는 문자열 `.snd`입니다.

sunau.**AUDIO_FILE_ENCODING_MULAW_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_16**
sunau.**AUDIO_FILE_ENCODING_LINEAR_24**
sunau.**AUDIO_FILE_ENCODING_LINEAR_32**
sunau.**AUDIO_FILE_ENCODING_ALAW_8**
: 이 모듈이 지원하는, AU 헤더의 인코딩 필드 값.

sunau.**AUDIO_FILE_ENCODING_FLOAT**
sunau.**AUDIO_FILE_ENCODING_DOUBLE**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G721**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G722**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_3**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_5**
추가로 알려진 AU 헤더의 인코딩 필드 값이지만, 이 모듈에서 지원하지 않는 값.

23.3.1 AU_read 객체

위의 *open()*에 의해 반환된 AU_read 객체는 다음과 같은 메서드를 가지고 있습니다:

AU_read.**close**()
스트림을 닫고, 인스턴스를 사용할 수 없게 합니다. (삭제 시 자동으로 호출됩니다.)

AU_read.**getnchannels**()
오디오 채널 수를 반환합니다 (모노는 1, 스테레오는 2).

AU_read.**getsampwidth**()
샘플 폭을 바이트 단위로 반환합니다.

AU_read.**getframerate**()
샘플링 빈도를 반환합니다.

AU_read.**getnframes**()
오디오 프레임의 수를 반환합니다.

AU_read.**getcomptype**()
압축 유형을 반환합니다. 지원되는 압축 유형은 `'ULAW'`, `'ALAW'` 및 `'NONE'`입니다.

AU_read.**getcompname**()
*getcomptype()*의 사람이 읽을 수 있는 버전. 지원되는 유형은 각각 `'CCITT G.711 u-law'`, `'CCITT G.711 A-law'` 및 `'not compressed'` 이름입니다.

AU_read.**getparams**()
`get*()` 메서드의 결과와 동등한, *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`를 반환합니다.

AU_read.**readframes**(*n*)
최대 *n* 프레임의 오디오를 *bytes* 객체로 읽고 반환합니다. 데이터는 선형 형식(linear format)으로 반환됩니다. 원본 데이터가 u-LAW 형식이면, 변환됩니다.

AU_read.**rewind**()
파일 포인터를 오디오 스트림의 시작 부분으로 되감습니다.

다음의 두 메서드는 이들 사이에서 호환 가능한 용어 "위치(position)"를 정의하며, 그 외에는 구현에 따라 다릅니다.

AU_read.**setpos**(*pos*)
파일 포인터를 지정된 위치로 설정합니다. *tell()*에서 반환된 값만 *pos*에 사용해야 합니다.

AU_read.**tell**()
현재 파일 포인터 위치를 반환합니다. 반환된 값은 파일에서의 실제 위치와 아무런 관련이 없음에 유의하십시오.

다음 두 함수는 *aifc*와의 호환성을 위해 정의되었으며, 흥미로운 작업을 수행하지 않습니다.

AU_read.**getmarkers**()
`None`을 반환합니다.

AU_read.**getmark**(*id*)
에러를 발생시킵니다.

23.3.2 AU_write 객체

위의 *open()* 에서 반환된 AU_write 객체에는 다음과 같은 메서드가 있습니다:

AU_write.**setnchannels**(*n*)
채널 수를 설정합니다.

AU_write.**setsampwidth**(*n*)
샘플 폭을 설정합니다 (바이트 단위).

버전 3.4에서 변경: 24비트 샘플에 대한 지원이 추가되었습니다.

AU_write.**setframerate**(*n*)
프레임 속도를 설정합니다.

AU_write.**setnframes**(*n*)
프레임 수를 설정합니다. 더 많은 프레임이 기록되면 나중에 변경될 수 있습니다.

AU_write.**setcomptype**(*type*, *name*)
압축 유형과 설명을 설정합니다. 출력에는 'NONE' 과 'ULAW' 만 지원됩니다.

AU_write.**setparams**(*tuple*)
*tuple*은 (nchannels, sampwidth, framerate, nframes, comptype, compname) 이어야 하며, set*() 메서드에 유효한 값이어야 합니다. 모든 파라미터를 설정합니다.

AU_write.**tell**()
파일의 현재 위치를 반환하는데, *AU_read.tell()* 과 *AU_read.setpos()* 메서드와 같은 면책 조항이 적용됩니다.

AU_write.**writeframesraw**(*data*)
*nframes*를 수정하지 않고 오디오 프레임을 씁니다.

버전 3.4에서 변경: 이제 모든 바이트열류 객체가 허락됩니다.

AU_write.**writeframes**(*data*)
오디오 프레임을 쓰고 *nframes*를 올바르게 만듭니다.

버전 3.4에서 변경: 이제 모든 바이트열류 객체가 허락됩니다.

AU_write.**close**()
*nframes*를 올바르게 만들고 파일을 닫습니다.

이 메서드는 삭제 시에 호출됩니다.

writeframes() 나 writeframesraw() 를 호출한 후 파라미터를 설정하는 것은 유효하지 않습니다.

23.4 wave — WAV 파일 읽고 쓰기

소스 코드: Lib/wave.py

---

wave 모듈은 WAV 음향 형식에 편리한 인터페이스를 제공합니다. 압축/압축 해제는 지원하지 않지만, 모노/스테레오를 지원합니다.

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.mode가 *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.

wave.**openfp**(*file*, *mode*)
*open()*의 동의어입니다. 이전 버전과의 호환성을 위해 유지됩니다.

Deprecated since version 3.7, will be removed in version 3.9.

exception wave.**Error**
WAV 명세를 위반하거나 구현 결함으로 인해 무언가가 불가능할 때 발생하는 에러.

23.4.1 Wave_read 객체

*open()*이 반환하는, Wave_read 객체는 다음과 같은 메서드를 가지고 있습니다:

Wave_read.**close**()
스트림이 *wave*에 의해 열렸다면 스트림을 닫고, 인스턴스를 사용할 수 없게 만듭니다. 이것은 객체가 가비지 수집될 때 자동으로 호출됩니다.

Wave_read.**getnchannels**()
오디오 채널 수를 반환합니다(모노는 1, 스테레오는 2).

Wave_read.**getsampwidth**()
샘플 폭을 바이트 단위로 반환합니다.

Wave_read.**getframerate**()
샘플링 빈도를 반환합니다.

Wave_read.**getnframes**()
오디오 프레임의 수를 반환합니다.

Wave_read.**getcomptype**()
압축 유형을 반환합니다(지원되는 유형은 'NONE' 뿐입니다).

Wave_read.**getcompname**()
*getcomptype()*의 사람이 읽을 수 있는 버전. 보통 'not compressed'이 'NONE'에 해당합니다.

Wave_read.**getparams**()
get*() 메서드의 결과와 동등한, *namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname)를 반환합니다.

Wave_read.**readframes**(*n*)
최대 *n* 프레임의 오디오를 *bytes* 객체로 읽고 반환합니다.

Wave_read.**rewind**()
파일 포인터를 오디오 스트림의 시작 부분으로 되감습니다.

다음의 두 메서드는 *aifc* 모듈과의 호환성을 위해 정의되었으며, 흥미로운 작업을 수행하지 않습니다.

Wave_read.**getmarkers**()
None을 반환합니다.

Wave_read.**getmark**(*id*)
에러를 발생시킵니다.

다음의 두 메서드는 이들 사이에서 호환 가능한 용어 "위치(position)"를 정의하며, 그 외에는 구현에 따라 다릅니다.

Wave_read.**setpos**(*pos*)
파일 포인터를 지정된 위치로 설정합니다.

Wave_read.**tell**()
현재 파일 포인터 위치를 반환합니다.

23.4.2 Wave_write 객체

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.

*open()*에 의해 반환된, Wave_write 객체에는 다음과 같은 메서드가 있습니다:

버전 3.4에서 변경: 위치 변경할 수 없는(unseekable) 파일에 대한 지원이 추가되었습니다.

Wave_write.**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.

Wave_write.**setnchannels**(*n*)
채널 수를 설정합니다.

Wave_write.**setsampwidth**(*n*)
샘플 폭을 *n* 바이트로 설정합니다.

Wave_write.**setframerate**(*n*)
프레임 속도를 *n*으로 설정합니다.

버전 3.2에서 변경: 이 메서드에 대한 비 정수 입력은 가장 가까운 정수로 자리 올림 됩니다.

Wave_write.**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).

Wave_write.**setcomptype**(*type*, *name*)
압축 유형과 설명을 설정합니다. 현재, 압축 유형 NONE 만 지원됩니다. 즉, 압축하지 않습니다.

Wave_write.**setparams**(*tuple*)
*tuple*은 (nchannels, sampwidth, framerate, nframes, comptype, compname)이어야 하며, set*() 메서드에 유효한 값이어야 합니다. 모든 파라미터를 설정합니다.

Wave_write.**tell**()
파일의 현재 위치를 반환하는데, *Wave_read.tell()* 과 *Wave_read.setpos()* 메서드와 같은 면책 조항이 적용됩니다.

Wave_write.**writeframesraw**(*data*)
*nframes*를 수정하지 않고 오디오 프레임을 씁니다.

버전 3.4에서 변경: 이제 모든 바이트열류 객체가 허락됩니다.

Wave_write.**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.

23.5 chunk — IFF 청크된 데이터 읽기

소스 코드: Lib/chunk.py

이 모듈은 EA IFF 85 청크를 사용하는 파일을 읽기 위한 인터페이스를 제공합니다.[1] 이 형식은 적어도 AIFF/AIFF-C (Audio Interchange File Format) 와 RMFF (Real Media File Format) 에서 사용됩니다. WAVE 오디오 파일 형식은 밀접하게 관련되어 있으며 이 모듈을 사용하여 읽을 수도 있습니다.

청크의 구조는 다음과 같습니다:

오프셋	길이	내용
0	4	청크 ID
4	4	빅 엔디안 바이트 순서로 청크의 크기. 헤더는 포함하지 않습니다.
8	*n*	데이터 바이트. 여기서 *n*은 앞 필드에서 주어진 크기입니다.
8 + *n*	0 또는 1	*n*가 홀수이고 청크 정렬이 사용된 경우 필요한 패드 바이트

ID는 청크의 유형을 식별하는 4바이트 문자열입니다.

크기 필드(빅 엔디안 바이트 순서를 사용하여 인코딩된 32비트 값)는 청크 데이터의 크기를 제공하며, 8바이트 헤더는 포함하지 않습니다.

일반적으로 IFF 형식의 파일은 하나 이상의 청크로 구성됩니다. 여기에 정의된 *Chunk* 클래스의 제안된 사용법은 각 청크의 시작 부분에서 인스턴스를 만들고 끝까지 도달할 때까지 인스턴스에서 읽는 것입니다. 그다음에 새 인스턴스를 만들 수 있습니다. 파일의 끝에서, 새 인스턴스를 만드는 것은 *EOFError* 예외로 실패합니다.

class chunk.**Chunk**(*file*, *align=True*, *bigendian=True*, *inclheader=False*)
청크를 나타내는 클래스. *file* 인자는 파일류 객체를 기대합니다. 이 클래스의 인스턴스가 특별히 허용됩니다. 필요한 유일한 메서드는 `read()` 입니다. *seek()* 와 *tell()* 메서드가 있고 예외를 발생시키지 않으면 이것들도 사용됩니다. 이러한 메서드가 존재하고, 예외가 발생하면, 객체가 변경되지 않았을 것으로 기대합니다. 선택적 인자 *align*이 참이면, 청크는 2바이트 경계에서 정렬되는 것으로 가정합니다. *align*이 거짓이면 정렬을 가정하지 않습니다. 기본값은 참입니다. 선택적 인자 *bigendian*이 거짓이면 청크 크기는 리틀 엔디안 순서로 간주합니다. 이것은 WAVE 오디오 파일에 필요합니다. 기본값은 참입니다. 선택적 인자 *inclheader*가 참이면, 청크 헤더에 주어진 크기는 헤더의 크기를 포함합니다. 기본값은 거짓입니다.

[1] "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, 1985년 1월.

Chunk 객체는 다음 메서드를 지원합니다:

getname()
청크의 이름(ID)을 돌려줍니다. 이것은 청크의 처음 4바이트입니다.

getsize()
청크의 크기를 돌려줍니다.

close()
닫고 청크의 끝으로 건너뜁니다. 하부 파일을 닫지 않습니다.

나머지 메서드는 *close()* 메서드가 호출된 후에 호출되면 *OSError*를 발생시킵니다. 파이썬 3.3 이전에는 *IOError*를 발생시켰습니다. 이제는 *OSError*의 별칭입니다.

isatty()
`False`를 반환합니다.

seek(*pos*, *whence=0*)
청크의 현재 위치를 설정합니다. *whence* 인자는 선택 사항이며 기본값은 `0`(절대 파일 위치 지정)입니다; 다른 값은 `1`(현재 위치에 상대적인 탐색)과 `2`(파일의 끝에 상대적인 탐색)입니다. 반환 값이 없습니다. 하부 파일이 탐색을 허용하지 않으면, 정방향 탐색만 허용됩니다.

tell()
청크의 현재 위치를 반환합니다.

read(*size=-1*)
청크에서 최대 *size* 바이트를 읽습니다 (*size* 바이트를 얻기 전에 read가 청크 끝에 도달하면 덜 읽을 수 있습니다). *size* 인자가 음수이거나 생략되면, 청크의 끝까지 모든 데이터를 읽습니다. 청크의 끝이 즉시 발견되면 빈 바이트열 객체가 반환됩니다.

skip()
청크의 끝으로 건너뜁니다. 청크에 대한 모든 추가 *read()* 호출은 `b''`를 반환합니다. 청크의 내용에 관심이 없으면, 파일이 다음 청크의 시작을 가리키도록 이 메서드를 호출해야 합니다.

23.6 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, I와 Q 좌표는 양수나 음수가 될 수 있습니다. 다른 모든 공간에서, 좌표는 모두 0과 1 사이입니다.

더 보기:

색 공간에 대한 자세한 내용은 http://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)
```

23.7 imghdr — 이미지 유형 판단

소스 코드: Lib/imghdr.py

---

imghdr 모듈은 파일이나 바이트 스트림에 포함된 이미지의 유형을 판단합니다.

imghdr 모듈은 다음 함수를 정의합니다:

imghdr.**what**(*filename*, *h=None*)
*filename*으로 이름이 지정된 파일에 포함된 이미지 데이터를 검사하고, 이미지 유형을 설명하는 문자열을 반환합니다. 선택적 *h*가 제공되면, *filename*는 무시되고 *h*가 검사할 바이트 스트림을 포함한다고 가정합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

아래에 *what()*의 반환 값과 함께 나열된 것처럼, 다음과 같은 이미지 유형을 인식합니다:

값	이미지 형식
'rgb'	SGI ImgLib 파일
'gif'	GIF 87a 과 89a 파일
'pbm'	Portable Bitmap 파일
'pgm'	Portable Graymap 파일
'ppm'	Portable Pixmap 파일
'tiff'	TIFF 파일
'rast'	Sun Raster 파일
'xbm'	X Bitmap 파일
'jpeg'	JFIF 나 Exif 형식의 JPEG 데이터
'bmp'	BMP 파일
'png'	Portable Network Graphics
'webp'	WebP 파일
'exr'	OpenEXR 파일

버전 3.5에 추가: *exr* 과 *webp* 형식이 추가되었습니다.

이 변수에 추가해서 *imghdr*가 인식할 수 있는 파일 유형 목록을 확장할 수 있습니다:

imghdr.**tests**
개별검사를 수행하는 함수 리스트. 각 함수는 두 개의 인자를 받아들입니다: 바이트 스트림과 열린 파일류 객체. *what()*이 바이트 스트림으로 호출되면, 파일류 객체는 None이 됩니다.

검사 함수는 검사가 성공하면 이미지 유형을 설명하는 문자열을 반환하고, 실패하면 None을 반환해야 합니다.

예제:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

23.8 sndhdr — 음향 파일 유형 판단

소스 코드: Lib/sndhdr.py

---

*sndhdr*은 파일에 있는 음향 데이터 유형을 판별하려고 하는 유틸리티 함수를 제공합니다. 이 함수는 파일에 저장된 음향 데이터의 형식을 결정할 수 있을 때 5가지 어트리뷰트를 포함하는 *namedtuple()*을 반환합니다: (filetype, framerate, nchannels, nframes, sampwidth). *type*의 값은 데이터 유형을 나타내며 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub' 또는 'ul' 문자열 중 하나가 됩니다. *sampling_rate*는 실제 값이거나 알 수 없거나 디코드하기 어려우면 0입니다. 마찬가지로, *channels*는 채널 수거나 결정할 수 없거나 값을 디코드하기 어려우면 0이 됩니다. *frames* 의 값은 프레임 수 또는 -1이 됩니다. 튜플의 마지막 항목인 *bits_per_sample*는 비트 단위의 샘플 크기이거나 A-LAW의 경우 'A' 또는 u-LAW의 경우 'U' 입니다.

sndhdr.**what**(*filename*)
*whathdr()*를 사용하여 *filename* 파일에 저장된 음향 데이터 유형을 판단합니다. 성공하면 위의 설명과 같이 네임드 튜플을 반환합니다. 그렇지 않으면 None을 반환합니다.

버전 3.5에서 변경: 결과가 튜플에서 네임드 튜플로 변경되었습니다.

sndhdr.**whathdr**(*filename*)
파일 헤더에 따라 파일에 저장된 음향 데이터의 유형을 판단합니다. 파일의 이름은 *filename*으로 주어집니다. 이 함수는 성공 시에 위에서 설명한 네임드 튜플을 반환하고, 그렇지 않으면 None을 반환합니다.

버전 3.5에서 변경: 결과가 튜플에서 네임드 튜플로 변경되었습니다.

23.9 ossaudiodev — Access to OSS-compatible audio devices

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

버전 3.3에서 변경: Operations in this module now raise *OSError* where *IOError* was raised.

더 보기:

Open Sound System Programmer's Guide the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see <sys/soundcard.h> on either Linux or FreeBSD for a listing.

ossaudiodev defines the following variables and functions:

exception ossaudiodev.**OSSAudioError**
This exception is raised on certain errors. The argument is a string describing what went wrong.

(If *ossaudiodev* receives an error from a system call such as `open()`, `write()`, or `ioctl()`, it raises *OSError*. Errors detected directly by *ossaudiodev* result in *OSSAudioError*.)

(For backwards compatibility, the exception class is also available as `ossaudiodev.error`.)

ossaudiodev.**open**(*mode*)
ossaudiodev.**open**(*device*, *mode*)
Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as `read()`, `write()`, and `fileno()` (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

device is the audio device filename to use. If it is not specified, this module first looks in the environment variable `AUDIODEV` for a device to use. If not found, it falls back to `/dev/dsp`.

mode is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older `linuxaudiodev` module which *ossaudiodev* supersedes.

ossaudiodev.**openmixer**([*device*])
Open a mixer device and return an OSS mixer device object. *device* is the mixer device filename to use. If it is not specified, this module first looks in the environment variable `MIXERDEV` for a device to use. If not found, it falls back to `/dev/mixer`.

23.9.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by *open()* define the following methods and (read-only) attributes:

oss_audio_device.**close**()
Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

oss_audio_device.**fileno**()
Return the file descriptor associated with the device.

oss_audio_device.**read**(*size*)
Read *size* bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block *read()* until the entire requested amount of data is available.

oss_audio_device.**write**(*data*)
Write a *bytes-like object* *data* to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire data is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written—see *writeall()*.

버전 3.5에서 변경: Writable *bytes-like object* is now accepted.

oss_audio_device.**writeall**(*data*)
Write a *bytes-like object* *data* to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as *write()*; *writeall()* is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

버전 3.5에서 변경: Writable *bytes-like object* is now accepted.

버전 3.2에서 변경: Audio device objects also support the context management protocol, i.e. they can be used in a with statement.

The following methods each map to exactly one ioctl() system call. The correspondence is obvious: for example, setfmt() corresponds to the SNDCTL_DSP_SETFMT ioctl, and sync() to SNDCTL_DSP_SYNC (this can be useful when consulting the OSS documentation). If the underlying ioctl() fails, they all raise *OSError*.

oss_audio_device.**nonblock**()
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

oss_audio_device.**getfmts**()
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

Format	Description
AFMT_MU_LAW	a logarithmic encoding (used by Sun .au files and /dev/audio)
AFMT_A_LAW	a logarithmic encoding
AFMT_IMA_ADPCM	a 4:1 compressed format defined by the Interactive Multimedia Association
AFMT_U8	Unsigned, 8-bit audio
AFMT_S16_LE	Signed, 16-bit audio, little-endian byte order (as used by Intel processors)
AFMT_S16_BE	Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
AFMT_S8	Signed, 8 bit audio
AFMT_U16_LE	Unsigned, 16-bit little-endian audio
AFMT_U16_BE	Unsigned, 16-bit big-endian audio

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support AFMT_U8; the most common format used today is AFMT_S16_LE.

oss_audio_device.**setfmt**(*format*)
Try to set the current audio format to *format*—see *getfmts()* for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an "audio format" of AFMT_QUERY.

oss_audio_device.**channels**(*nchannels*)
Set the number of output channels to *nchannels*. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device.**speed**(*samplerate*)
Try to set the audio sampling rate to *samplerate* samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are:

Rate	Description
8000	default rate for /dev/audio
11025	speech recording
22050	
44100	CD quality audio (at 16 bits/sample and 2 channels)
96000	DVD quality audio (at 24 bits/sample)

oss_audio_device.**sync**()
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using *sync()*.

oss_audio_device.**reset**()
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling *reset()*.

oss_audio_device.**post**()
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.**setparameters**(*format*, *nchannels*, *samplerate*[, *strict=False*])
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format*, *nchannels*, and *samplerate* should be as specified in the *setfmt()*, *channels()*, and *speed()* methods. If *strict* is true, *setparameters()* checks to see if each parameter was actually set to the requested value, and raises *OSSAudioError* if not. Returns a tuple (*format*, *nchannels*, *samplerate*) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of *setfmt()*, *channels()*, and *speed()*).

For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.**bufsize**()
Returns the size of the hardware buffer, in samples.

oss_audio_device.**obufcount**()
Returns the number of samples that are in the hardware buffer yet to be played.

oss_audio_device.**obuffree**()
Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

oss_audio_device.**closed**
Boolean indicating whether the device has been closed.

oss_audio_device.**name**
String containing the name of the device file.

oss_audio_device.**mode**
The I/O mode for the file, either "r", "rw", or "w".

23.9.2 Mixer Device Objects

The mixer object provides two file-like methods:

oss_mixer_device.**close**()
This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an *OSError*.

oss_mixer_device.**fileno**()
Returns the file handle number of the open mixer device file.

버전 3.2에서 변경: Mixer objects also support the context management protocol.

The remaining methods are specific to audio mixing:

oss_mixer_device.**controls**()
This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a subset of all available mixer controls—the SOUND_MIXER_* constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

oss_mixer_device.**stereocontrols**()
Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with *controls()* to determine which).

See the code example for the *controls()* function for an example of getting data from a bitmask.

oss_mixer_device.**reccontrols**()
Returns a bitmask specifying the mixer controls that may be used to record. See the code example for *controls()* for an example of reading from a bitmask.

oss_mixer_device.**get**(*control*)
Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises *OSSAudioError* if an invalid control is specified, or *OSError* if an unsupported control is specified.

oss_mixer_device.**set**(*control*, *(left*, *right)*)
Sets the volume for a given mixer control to (left,right). left and right must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises *OSSAudioError* if an invalid mixer control was specified, or if the specified volumes were out-of-range.

oss_mixer_device.**get_recsrc**()
This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.**set_recsrc**(*bitmask*)
Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources)

if successful; raises *OSError* if an invalid source was specified. To set the current recording source to the microphone input:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

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` — Multilingual internationalization services

Source code: Lib/gettext.py

The *gettext* module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU **gettext** message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

24.1.1 GNU `gettext` API

The *gettext* module defines the following API, which is very similar to the GNU **gettext** API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

gettext.**bindtextdomain**(*domain*, *localedir=None*)

Bind the *domain* to the locale directory *localedir*. More concretely, *gettext* will look for binary `.mo` files for the given domain using the path (on Unix): *localedir*/*language*/LC_MESSAGES/*domain*.mo, where *languages* is searched for in the environment variables LANGUAGE, LC_ALL, LC_MESSAGES, and LANG respectively.

If *localedir* is omitted or `None`, then the current binding for *domain* is returned.[1]

`gettext.`**`bind_textdomain_codeset`**(*domain*, *codeset=None*)
Bind the *domain* to *codeset*, changing the encoding of byte strings returned by the *lgettext()*, *ldgettext()*, *lngettext()* and *ldngettext()* functions. If *codeset* is omitted, then the current binding is returned.

`gettext.`**`textdomain`**(*domain=None*)
Change or query the current global domain. If *domain* is `None`, then the current global domain is returned, otherwise the global domain is set to *domain*, which is returned.

`gettext.`**`gettext`**(*message*)
Return the localized translation of *message*, based on the current global domain, language, and locale directory. This function is usually aliased as `_()` in the local namespace (see examples below).

`gettext.`**`dgettext`**(*domain*, *message*)
Like *gettext()*, but look the message up in the specified *domain*.

`gettext.`**`ngettext`**(*singular*, *plural*, *n*)
Like *gettext()*, but consider plural forms. If a translation is found, apply the plural formula to *n*, and return the resulting message (some languages have more than two plural forms). If no translation is found, return *singular* if *n* is 1; return *plural* otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable *n*; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in `.po` files and the formulas for a variety of languages.

`gettext.`**`dngettext`**(*domain*, *singular*, *plural*, *n*)
Like *ngettext()*, but look the message up in the specified *domain*.

`gettext.`**`lgettext`**(*message*)

`gettext.`**`ldgettext`**(*domain*, *message*)

`gettext.`**`lngettext`**(*singular*, *plural*, *n*)

`gettext.`**`ldngettext`**(*domain*, *singular*, *plural*, *n*)
Equivalent to the corresponding functions without the `l` prefix (*gettext()*, *dgettext()*, *ngettext()* and *dngettext()*), but the translation is returned as a byte string encoded in the preferred system encoding if no other encoding was explicitly set with *bind_textdomain_codeset()*.

> **경고:** These functions should be avoided in Python 3, because they return encoded bytes. It's much better to use alternatives which return Unicode strings instead, since most Python applications will want to manipulate human readable text as strings instead of bytes. Further, it's possible that you may get unexpected Unicode-related exceptions if there are encoding problems with the translated strings. It is possible that the `l*()` functions will be deprecated in future Python versions due to their inherent problems and limitations.

Note that GNU **gettext** also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
```

(다음 페이지에 계속)

[1] The default locale directory is system dependent; for example, on RedHat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The *gettext* module does not try to support these system dependent defaults; instead its default is *sys.base_prefix*`/share/locale` (see *sys.base_prefix*). For this reason, it is always best to call *bindtextdomain()* with an explicit absolute path at the start of your application.

(이전 페이지에서 계속)

```
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

24.1.2 Class-based API

The class-based API of the *gettext* module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a *GNUTranslations* class which implements the parsing of GNU `.mo` format files, and has methods for returning strings. Instances of this class can also install themselves in the built-in namespace as the function `_()`.

`gettext.`**`find`**(*domain*, *localedir=None*, *languages=None*, *all=False*)

This function implements the standard `.mo` file search algorithm. It takes a *domain*, identical to what *textdomain()* takes. Optional *localedir* is as in *bindtextdomain()*. Optional *languages* is a list of strings, where each string is a language code.

If *localedir* is not given, then the default system locale directory is used.[2] If *languages* is not given, then the following environment variables are searched: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`. The first one returning a non-empty value is used for the *languages* variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

find() then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

localedir`/`*language*`/LC_MESSAGES/`*domain*`.mo`

The first such file name that exists is returned by *find()*. If no such file is found, then `None` is returned. If *all* is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

`gettext.`**`translation`**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*, *codeset=None*)

Return a `*Translations` instance based on the *domain*, *localedir*, and *languages*, which are first passed to *find()* to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is *class_* if provided, otherwise *GNUTranslations*. The class's constructor must take a single *file object* argument. If provided, *codeset* will change the charset used to encode translated strings in the *lgettext()* and *lngettext()* methods.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, *copy.copy()* is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no `.mo` file is found, this function raises *OSError* if *fallback* is false (which is the default), and returns a *NullTranslations* instance if *fallback* is true.

버전 3.3에서 변경: *IOError* used to be raised instead of *OSError*.

`gettext.`**`install`**(*domain*, *localedir=None*, *codeset=None*, *names=None*)

This installs the function `_()` in Python's builtins namespace, based on *domain*, *localedir*, and *codeset* which are passed to the function *translation()*.

For the *names* parameter, please see the description of the translation object's *install()* method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the `_()` function, like this:

[2] See the footnote for *bindtextdomain()* above.

```
print(_('This string will be translated.'))
```

For convenience, you want the `_()` function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

The `NullTranslations` class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is *NullTranslations*; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of `NullTranslations`:

class gettext.**NullTranslations**(*fp=None*)

Takes an optional *file object* *fp*, which is ignored by the base class. Initializes "protected" instance variables *_info* and *_charset* which are set by derived classes, as well as *_fallback*, which is set through *add_fallback()*. It then calls `self._parse(fp)` if *fp* is not `None`.

_parse(*fp*)

No-op in the base class, this method takes file object *fp*, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(*fallback*)

Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext(*message*)

If a fallback has been set, forward `gettext()` to the fallback. Otherwise, return *message*. Overridden in derived classes.

ngettext(*singular*, *plural*, *n*)

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return *singular* if *n* is 1; return *plural* otherwise. Overridden in derived classes.

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

Equivalent to *gettext()* and *ngettext()*, but the translation is returned as a byte string encoded in the preferred system encoding if no encoding was explicitly set with *set_output_charset()*. Overridden in derived classes.

> **경고:** These methods should be avoided in Python 3. See the warning for the *lgettext()* function.

info()

Return the "protected" `_info` variable, a dictionary containing the metadata found in the message catalog file.

charset()

Return the encoding of the message catalog file.

output_charset()

Return the encoding used to return translated messages in *lgettext()* and *lngettext()*.

set_output_charset(*charset*)

Change the encoding used to return translated messages.

install(*names=None*)

This method installs *gettext()* into the built-in namespace, binding it to _.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to `_()`. Supported names are `'gettext'`, `'ngettext'`, `'lgettext'` and `'lngettext'`.

Note that this is only one way, albeit the most convenient way, to make the `_()` function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install `_()`. Instead, they should use this code to make `_()` available to their module:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts `_()` only in the module's global namespace and so only affects calls within this module.

The `GNUTranslations` class

The *gettext* module provides one additional class derived from *NullTranslations*: *GNUTranslations*. This class overrides `_parse()` to enable reading GNU **gettext** format `.mo` files in both big-endian and little-endian format.

GNUTranslations parses optional metadata out of the translation catalog. It is convention with GNU **gettext** to include metadata as the translation for the empty string. This metadata is in **RFC 822**-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the "protected" `_charset` instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII is assumed.

Since message ids are read as Unicode strings too, all `*gettext()` methods will assume message ids as Unicode strings, not byte strings.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" `_info` instance variable.

If the `.mo` file's magic number is invalid, the major version number is unexpected, or if other problems occur while reading the file, instantiating a *GNUTranslations* class can raise *OSError*.

class gettext.**GNUTranslations**

The following methods are overridden from the base class implementation:

gettext(*message*)

Look up the *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's *gettext()* method. Otherwise, the *message* id is returned.

ngettext(*singular*, *plural*, *n*)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's *ngettext()* method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Here is an example:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

Equivalent to *gettext()* and *ngettext()*, but the translation is returned as a byte string encoded in the preferred system encoding if no encoding was explicitly set with *set_output_charset()*.

> **경고:** These methods should be avoided in Python 3. See the warning for the *lgettext()* function.

Solaris message catalog support

The Solaris operating system defines its own binary `.mo` file format, but since no documentation can be found on this format, it is not supported at this time.

The Catalog constructor

GNOME uses a version of the *gettext* module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

For compatibility with this older module, the function `Catalog()` is an alias for the *translation()* function described above.

One difference between this module and Henstridge's: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

24.1.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language-specific translations of the message catalogs
4. use the *gettext* module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` — that is, a call to the function `_()`. For example:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

In this example, the string `'writing a log message'` is marked as a candidate for translation, while the strings `'mylog.txt'` and `'w'` are not.

There are a few tools to extract the strings meant for translation. The original GNU **gettext** only supported C or C++ source code but its extended version **xgettext** scans code written in a number of languages, including Python, to find strings marked as translatable. Babel is a Python internationalization library that includes a `pybabel` script to extract and compile message catalogs. François Pinard's program called **xpot** does a similar job and is available as part of his po-utils package.

(Python also includes pure-Python versions of these programs, called **pygettext.py** and **msgfmt.py**; some Python distributions will install them for you. **pygettext.py** is similar to **xgettext**, but only understands Python source code and cannot handle other programming languages such as C or C++. **pygettext.py** supports a command-line interface similar to **xgettext**; for details on its use, run `pygettext.py --help`. **msgfmt.py** is binary compatible with GNU **msgfmt**. With these two programs, you may not need the GNU **gettext** package to internationalize your Python applications.)

xgettext, **pygettext**, and similar tools generate `.po` files that are message catalogs. They are structured human-readable files that contain every marked string in the source code, along with a placeholder for the translated versions of these strings.

Copies of these `.po` files are then handed over to the individual human translators who write translations for every supported natural language. They send back the completed language-specific versions as a `<language-name>.po` file that's compiled into a machine-readable `.mo` binary catalog file using the **msgfmt** program. The `.mo` files are used by the *gettext* module for the actual translation processing at run-time.

How you use the *gettext* module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU **gettext** API but instead the class-based API.

Let's say your module is called "spam" and the module's various natural language translation `.mo` files reside in `/usr/share/locale` in GNU **gettext** format. Here's what you would put at the top of your module:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory, you can pass it into the *install()* function:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don't actually want to translate them until they are printed.

Here is one way you can handle this situation:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

This works because the dummy definition of `_()` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_()` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_()` in the local namespace.

Note that the second use of `_()` will not identify "a" as being translatable to the **gettext** program, because the parameter is not a string literal.

Another way to handle this is with the following example:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

In this case, you are marking translatable strings with the function `N_()`, which won't conflict with any definition of `_()`. However, you will need to teach your message extraction program to look for translatable strings marked with `N_()`. **xgettext**, **pygettext**, `pybabel extract`, and **xpot** all support this through the use of the `-k` command-line switch. The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

24.1.4 Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

24.2 `locale` — Internationalization services

Source code: Lib/locale.py

The *locale* module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The *locale* module is implemented on top of the `_locale` module, which in turn uses an ANSI C locale implementation if available.

The *locale* module defines the following exception and functions:

exception locale.**Error**
Exception raised when the locale passed to *setlocale()* is not recognized.

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 an iterable of two strings (language code and encoding). If it's an iterable, it's converted to a locale name using the locale aliasing engine. 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 omitted or None, the current setting for *category* is returned.

setlocale() is not thread-safe on most systems. Applications typically start with a call of

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user's default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

locale.**localeconv**()
Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

Category	Key	Meaning
LC_NUMERIC	'decimal_point'	Decimal point character.
	'grouping'	Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. If the sequence is terminated with *CHAR_MAX*, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly used.
	'thousands_sep'	Character used between groups.
LC_MONETARY	'int_curr_symbol'	International currency symbol.
	'currency_symbol'	Local currency symbol.
	'p_cs_precedes/n_cs_precedes'	Whether the currency symbol precedes the value (for positive resp. negative values).
	'p_sep_by_space/n_sep_by_space'	Whether the currency symbol is separated from the value by a space (for positive resp. negative values).
	'mon_decimal_point'	Decimal point used for monetary values.
	'frac_digits'	Number of fractional digits used in local formatting of monetary values.
	'int_frac_digits'	Number of fractional digits used in international formatting of monetary values.
	'mon_thousands_sep'	Group separator used for monetary values.
	'mon_grouping'	Equivalent to 'grouping', used for monetary values.
	'positive_sign'	Symbol used to annotate a positive monetary value.
	'negative_sign'	Symbol used to annotate a negative monetary value.
	'p_sign_posn/n_sign_posn'	The position of the sign (for positive resp. negative values), see below.

All numeric values can be set to *CHAR_MAX* to indicate that there is no value specified in this locale.

The possible values for 'p_sign_posn' and 'n_sign_posn' are given below.

Value	Explanation
0	Currency and value are surrounded by parentheses.
1	The sign should precede the value and currency symbol.
2	The sign should follow the value and currency symbol.
3	The sign should immediately precede the value.
4	The sign should immediately follow the value.
CHAR_MAX	Nothing is specified in this locale.

The function sets temporarily the LC_CTYPE locale to the LC_NUMERIC locale or the LC_MONETARY locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

버전 3.7에서 변경: The function now sets temporarily the LC_CTYPE locale to the LC_NUMERIC locale in

some cases.

locale.**nl_langinfo**(*option*)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The *nl_langinfo()* function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

locale.**CODESET**

Get a string with the name of the character encoding used in the selected locale.

locale.**D_T_FMT**

Get a string that can be used as a format string for *time.strftime()* to represent date and time in a locale-specific way.

locale.**D_FMT**

Get a string that can be used as a format string for *time.strftime()* to represent a date in a locale-specific way.

locale.**T_FMT**

Get a string that can be used as a format string for *time.strftime()* to represent a time in a locale-specific way.

locale.**T_FMT_AMPM**

Get a format string for *time.strftime()* to represent time in the am/pm format.

DAY_1 ... DAY_7

Get the name of the n-th day of the week.

참고: This follows the US convention of DAY_1 being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

ABDAY_1 ... ABDAY_7

Get the abbreviated name of the n-th day of the week.

MON_1 ... MON_12

Get the name of the n-th month.

ABMON_1 ... ABMON_12

Get the abbreviated name of the n-th month.

locale.**RADIXCHAR**

Get the radix character (decimal dot, decimal comma, etc.).

locale.**THOUSEP**

Get the separator character for thousands (groups of three digits).

locale.**YESEXPR**

Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

참고: The expression is in the syntax suitable for the regex() function from the C library, which might differ from the syntax used in *re*.

---

locale.**NOEXPR**
Get a regular expression that can be used with the regex(3) function to recognize a negative response to a yes/no question.

locale.**CRNCYSTR**
Get the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

locale.**ERA**
Get a string that represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the *time.strftime()* function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

locale.**ERA_D_T_FMT**
Get a format string for *time.strftime()* to represent date and time in a locale-specific era-based way.

locale.**ERA_D_FMT**
Get a format string for *time.strftime()* to represent a date in a locale-specific era-based way.

locale.**ERA_T_FMT**
Get a format string for *time.strftime()* to represent a time in a locale-specific era-based way.

locale.**ALT_DIGITS**
Get a representation of up to 100 values used to represent the values 0 to 99.

locale.**getdefaultlocale**([*envvars*])
Tries to determine the default locale settings and returns them as a tuple of the form `(language code, encoding)`.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable `'C'` locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the `LANG` variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the `LANG` variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. *envvars* defaults to the search path used in GNU gettext; it must always contain the variable name `'LANG'`. The GNU gettext search path contains `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` and `'LANGUAGE'`, in that order.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

locale.**getlocale**(*category=LC_CTYPE*)
Returns the current setting for the given locale category as sequence containing *language code*, *encoding*. *category* may be one of the `LC_*` values except *LC_ALL*. It defaults to *LC_CTYPE*.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

locale.**getpreferredencoding**(*do_setlocale=True*)
Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke *setlocale()* to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to `False`.

On Android or in the UTF-8 mode (`-X utf8` option), always return `'UTF-8'`, the locale and the *do_setlocale* argument are ignored.

버전 3.7에서 변경: The function now always returns `UTF-8` on Android or if the UTF-8 mode is enabled.

`locale.`**`normalize`**(*localename*)
Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with *setlocale()*. If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like *setlocale()*.

`locale.`**`resetlocale`**(*category=LC_ALL*)
Sets the locale for *category* to the default setting.

The default setting is determined by calling *getdefaultlocale()*. *category* defaults to *LC_ALL*.

`locale.`**`strcoll`**(*string1*, *string2*)
Compares two strings according to the current *LC_COLLATE* setting. As any other compare function, returns a negative, or a positive value, or `0`, depending on whether *string1* collates before or after *string2* or is equal to it.

`locale.`**`strxfrm`**(*string*)
Transforms a string to one that can be used in locale-aware comparisons. For example, `strxfrm(s1) < strxfrm(s2)` is equivalent to `strcoll(s1, s2) < 0`. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

`locale.`**`format_string`**(*format*, *val*, *grouping=False*, *monetary=False*)
Formats a number *val* according to the current *LC_NUMERIC* setting. The format follows the conventions of the `%` operator. For floating point values, the decimal point is modified if appropriate. If *grouping* is true, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

버전 3.7에서 변경: The *monetary* keyword parameter was added.

`locale.`**`format`**(*format*, *val*, *grouping=False*, *monetary=False*)
Please note that this function works like *format_string()* but will only work for exactly one `%char` specifier. For example, `'%f'` and `'%.0f'` are both valid specifiers, but `'%f KiB'` is not.

For whole format strings, use *format_string()*.

버전 3.7부터 폐지: Use *format_string()* instead.

`locale.`**`currency`**(*val*, *symbol=True*, *grouping=False*, *international=False*)
Formats a number *val* according to the current *LC_MONETARY* settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is true (which is not the default), grouping is done with the value. If *international* is true (which is not the default), the international currency symbol is used.

Note that this function will not work with the 'C' locale, so you have to set a locale via *setlocale()* first.

`locale.`**`str`**(*float*)
Formats a floating point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

`locale.`**`delocalize`**(*string*)
Converts a string into a normalized number string, following the *LC_NUMERIC* settings.

버전 3.5에 추가.

locale.**atof**(*string*)
Converts a string to a floating point number, following the *LC_NUMERIC* settings.

locale.**atoi**(*string*)
Converts a string to an integer, following the *LC_NUMERIC* conventions.

locale.**LC_CTYPE**
Locale category for the character type functions. Depending on the settings of this category, the functions of module *string* dealing with case change their behaviour.

locale.**LC_COLLATE**
Locale category for sorting strings. The functions *strcoll()* and *strxfrm()* of the *locale* module are affected.

locale.**LC_TIME**
Locale category for the formatting of time. The function *time.strftime()* follows these conventions.

locale.**LC_MONETARY**
Locale category for formatting of monetary values. The available options are available from the *localeconv()* function.

locale.**LC_MESSAGES**
Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by *os.strerror()* might be affected by this category.

locale.**LC_NUMERIC**
Locale category for formatting numbers. The functions *format()*, *atoi()*, *atof()* and *str()* of the *locale* module are affected by that category. All other numeric formatting operations are not affected.

locale.**LC_ALL**
Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.**CHAR_MAX**
This is a symbolic constant used for different values returned by *localeconv()*.

Example:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')   # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved locale
```

24.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the `C` locale, no matter what the user's preferred locale is. There is one exception: the *LC_CTYPE* category is changed at startup to set the current locale encoding to the user's preferred locale encoding. The program must explicitly say that it wants the user's preferred locale settings for other categories by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call *setlocale()* in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with *time.strftime()*), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-`C` locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: *atof()*, *atoi()*, *format()*, *str()*.

There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.

24.2.2 For extension writers and programs that embed Python

Extension modules should never call *setlocale()*, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is `C`).

When Python code uses the *locale* module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the `_locale` extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the `_locale` module is not accessible as a shared library.

24.2.3 Access to message catalogs

`locale.`**`gettext`**(*msg*)

`locale.`**`dgettext`**(*domain*, *msg*)

`locale.`**`dcgettext`**(*domain*, *msg*, *category*)

`locale.`**`textdomain`**(*domain*)

`locale.`**`bindtextdomain`**(*domain*, *dir*)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions `gettext()`, `dgettext()`, `dcgettext()`, `textdomain()`, `bindtextdomain()`, and `bind_textdomain_codeset()`. These are similar to the same functions in the *gettext* module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use *gettext* instead. A known exception to this rule are applications that link with additional C libraries which internally invoke `gettext()` or

`dcgettext()`. For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

CHAPTER 25

프로그램 프레임워크

이 장에서 설명하는 모듈은 주로 프로그램의 구조를 결정하는 프레임워크입니다. 현재 여기에 설명된 모듈은 모두 명령 줄 인터페이스 작성을 지향합니다.

이 장에서 설명하는 모듈의 전체 목록은 다음과 같습니다:

25.1 `turtle` — Turtle graphics

Source code: Lib/turtle.py

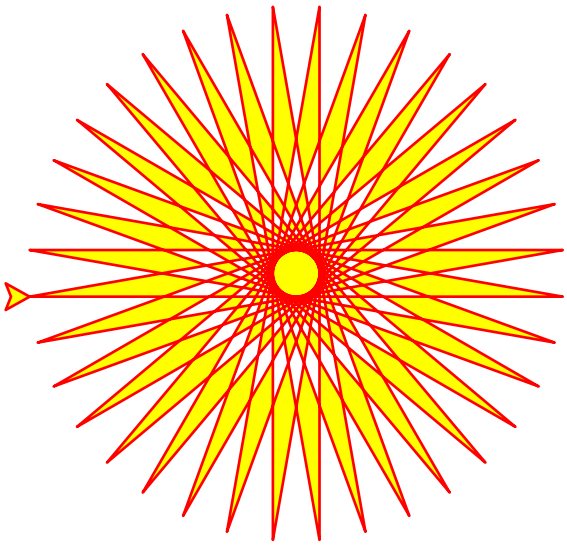
25.1.1 Introduction

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an `import turtle`, give it the command `turtle.forward(15)`, and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command `turtle.right(25)`, and it rotates in-place 25 degrees clockwise.

Turtle star

Turtle can draw intricate shapes using programs that repeat simple moves.



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

The *turtle* module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module and to be (nearly) 100% compatible with it. This means in the first place to enable the learning programmer to use all the commands, classes and methods interactively when using the module from within IDLE run with the `-n` switch.

The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses *tkinter* for the underlying graphics, it needs a version of Python installed with Tk support.

The object-oriented interface uses essentially two+two classes:

1. The *TurtleScreen* class defines graphics windows as a playground for the drawing turtles. Its constructor needs a `tkinter.Canvas` or a *ScrolledCanvas* as argument. It should be used when *turtle* is used as part of some application.

   The function *Screen()* returns a singleton object of a *TurtleScreen* subclass. This function should be used when *turtle* is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

   All methods of TurtleScreen/Screen also exist as functions, i.e. as part of the procedure-oriented interface.

2. *RawTurtle* (alias: *RawPen*) defines Turtle objects which draw on a *TurtleScreen*. Its constructor needs a Canvas, ScrolledCanvas or TurtleScreen as argument, so the RawTurtle objects know where to draw.

Derived from RawTurtle is the subclass *Turtle* (alias: Pen), which draws on "the" *Screen* instance which is automatically created, if not already present.

All methods of RawTurtle/Turtle also exist as functions, i.e. part of the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes *Screen* and *Turtle*. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a Screen method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen one has to use the object-oriented interface.

참고: In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument *self* which is omitted here.

25.1.2 Overview of available Turtle and Screen methods

Turtle methods

Turtle motion

Move and draw

forward() | *fd()*
backward() | *bk()* | *back()*
right() | *rt()*
left() | *lt()*
goto() | *setpos()* | *setposition()*
setx()
sety()
setheading() | *seth()*
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

Tell Turtle's state

position() | *pos()*
towards()
xcor()
ycor()
heading()
distance()

Setting and measurement

degrees()
radians()

Pen control

Drawing state

pendown() | *pd()* | *down()*
penup() | *pu()* | *up()*
pensize() | *width()*
pen()
isdown()

Color control

color()
pencolor()
fillcolor()

Filling

filling()
begin_fill()
end_fill()

More drawing control

reset()
clear()
write()

Turtle state

Visibility

showturtle() | *st()*
hideturtle() | *ht()*
isvisible()

Appearance

shape()
resizemode()
shapesize() | *turtlesize()*
shearfactor()
settiltangle()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

Using events

onclick()
onrelease()
ondrag()

Special Turtle methods

begin_poly()
end_poly()
get_poly()

clone()
getturtle() | *getpen()*
getscreen()
setundobuffer()
undobufferentries()

Methods of TurtleScreen/Screen

Window control

bgcolor()
bgpic()
clear() | *clearscreen()*
reset() | *resetscreen()*
screensize()
setworldcoordinates()

Animation control

delay()
tracer()
update()

Using screen events

listen()
onkey() | *onkeyrelease()*
onkeypress()
onclick() | *onscreenclick()*
ontimer()
mainloop() | *done()*

Settings and special methods

mode()
colormode()
getcanvas()
getshapes()
register_shape() | *addshape()*
turtles()
window_height()
window_width()

Input methods

textinput()
numinput()

Methods specific to Screen

bye()
exitonclick()
setup()
title()

25.1.3 Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called `turtle`.

Turtle motion

`turtle.`**`forward`**(*distance*)
`turtle.`**`fd`**(*distance*)

매개변수 **`distance`** – a number (integer or float)

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> 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`** – a number

Move the turtle backward by *distance*, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.`**`right`**(*angle*)
`turtle.`**`rt`**(*angle*)

매개변수 **`angle`** – a number (integer or float)

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the *degrees()* and *radians()* functions.) Angle orientation depends on the turtle mode, see *mode()*.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.`**`left`**(*angle*)
`turtle.`**`lt`**(*angle*)

매개변수 **`angle`** – a number (integer or float)

Turn turtle left by *angle* units. (Units are by default degrees, but can be set via the *degrees()* and *radians()* functions.) Angle orientation depends on the turtle mode, see *mode()*.

```
>>> 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** – a number or a pair/vector of numbers
- **y** – a number or None

If *y* is None, *x* must be a pair of coordinates or a *Vec2D* (e.g. as returned by *pos()*).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.

```
>>> 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.**setx**(*x*)

매개변수 **x** – a number (integer or float)

Set the turtle's first coordinate to *x*, leave second coordinate unchanged.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

매개변수 **y** – a number (integer or float)

Set the turtle's second coordinate to *y*, leave first coordinate unchanged.

```
>>> 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** – a number (integer or float)

Set the orientation of the turtle to *to_angle*. Here are some common directions in degrees:

standard mode	logo mode
0 - east	0 - north
90 - north	90 - east
180 - west	180 - south
270 - south	270 - west

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see *mode()*).

```
>>> 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** – a number
- **extent** – a number (or None)
- **steps** – an integer (or None)

Draw a circle with given *radius*. The center is *radius* units left of the turtle; *extent* – an angle – determines which part of the circle is drawn. If *extent* is not given, draw the entire circle. If *extent* is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if *radius* is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of *extent*.

As the circle is approximated by an inscribed regular polygon, *steps* determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

```
>>> 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)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

매개변수

- **size** – an integer >= 1 (if given)
- **color** – a colorstring or a numeric color tuple

Draw a circular dot with diameter *size*, using *color*. If *size* is not given, the maximum of pensize+4 and 2*pensize is used.

```
>>> 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**()

Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

매개변수 **stampid** – an integer, must be return value of previous *stamp()* call

Delete stamp with given *stampid*.

```
>>> 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** – an integer (or `None`)

Delete all or first/last *n* of turtle's stamps. If *n* is `None`, delete all stamps, if *n* > 0 delete first *n* stamps, else if *n* < 0 delete last *n* stamps.

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

매개변수 **speed** – an integer in the range 0..10 or a speedstring (see below)

Set the turtle's speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention: *speed* = 0 means that *no* animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Tell Turtle's state

turtle.**position**()
turtle.**pos**()
Return the turtle's current location (x,y) (as a *Vec2D* vector).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

매개변수

- **x** – a number or a pair/vector of numbers or a turtle instance
- **y** – a number if *x* is a number, else None

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - "standard"/"world" or "logo").

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()
Return the turtle's x coordinate.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()
Return the turtle's y coordinate.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()
Return the turtle's current heading (value depends on the turtle mode, see *mode()*).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

매개변수

- **x** – a number or a pair/vector of numbers or a turtle instance

- **y** – a number if *x* is a number, else `None`

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Settings for measurement

`turtle.`**`degrees`**(*fullcircle=360.0*)

매개변수 **`fullcircle`** – a number

Set angle measurement units, i.e. set number of "degrees" for a full circle. Default value is 360 degrees.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

`turtle.`**`radians`**()

Set the angle measurement units to radians. Equivalent to `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen control

Drawing state

turtle.**pendown**()
turtle.**pd**()
turtle.**down**()
Pull the pen down – drawing when moving.

turtle.**penup**()
turtle.**pu**()
turtle.**up**()
Pull the pen up – no drawing when moving.

turtle.**pensize**(*width=None*)
turtle.**width**(*width=None*)

매개변수 **width** – a positive number

Set the line thickness to *width* or return it. If resizemode is set to "auto" and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.**pen**(*pen=None*, ***pendict*)

매개변수

- **pen** – a dictionary with some or all of the below listed keys
- **pendict** – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen's attributes in a "pen-dictionary" with the following key/value pairs:

- "shown": True/False
- "pendown": True/False
- "pencolor": color-string or color-tuple
- "fillcolor": color-string or color-tuple
- "pensize": positive number
- "speed": number in range 0..10
- "resizemode": "auto" or "user" or "noresize"
- "stretchfactor": (positive number, positive number)
- "outline": positive number
- "tilt": number

This dictionary can be used as argument for a subsequent call to *pen()* to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 ('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**()

Return `True` if pen is down, `False` if it's up.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Color control

turtle.**pencolor**(**args*)

Return or set the pencolor.

Four input formats are allowed:

pencolor() Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

pencolor(colorstring) Set pencolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

pencolor((r, g, b)) Set pencolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

pencolor(r, g, b)

Set pencolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```
>>> colormode()
1.0
>>> turtle.pencolor()
'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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

Return or set the fillcolor.

Four input formats are allowed:

fillcolor() Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring) Set fillcolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

fillcolor((r, g, b)) Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

fillcolor(r, g, b)

Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color() Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by *pencolor()* and *fillcolor()*.

color(colorstring), color((r,g,b)), color(r,g,b) Inputs as in *pencolor()*, set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))

Equivalent to `pencolor(colorstring1)` and `fillcolor(colorstring2)` and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

See also: Screen method *colormode()*.

Filling

turtle.**filling**()
Return fillstate (`True` if filling, `False` else).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**begin_fill**()
To be called just before drawing a shape to be filled.

turtle.**end_fill**()
Fill the shape drawn after the last call to *begin_fill()*.

Whether or not overlap regions for self-intersecting polygons or multiple shapes are filled depends on the operating system graphics, type of overlap, and number of overlaps. For example, the Turtle star above may be either all yellow or have some white regions.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

More drawing control

turtle.**reset**()
Delete the turtle's drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> 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**()
Delete the turtle's drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.**write**(*arg*, *move=False*, *align="left"*, *font=("Arial", 8, "normal")*)

매개변수

- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – one of the strings “left”, “center” or right”
- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of *arg* - at the current turtle position according to *align* (“left”, “center” or right”) and with the given font. If *move* is true, the pen is moved to the bottom-right corner of the text. By default, *move* is `False`.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Turtle state

Visibility

`turtle.`**`hideturtle`**`()`
`turtle.`**`ht`**`()`
Make the turtle invisible. It’s a good idea to do this while you’re in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```
>>> turtle.hideturtle()
```

`turtle.`**`showturtle`**`()`
`turtle.`**`st`**`()`
Make the turtle visible.

```
>>> turtle.showturtle()
```

`turtle.`**`isvisible`**`()`
Return `True` if the Turtle is shown, `False` if it’s hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Appearance

`turtle.`**`shape`**(*name=None*)

매개변수 **name** – a string which is a valid shapename

Set turtle shape to shape with given *name* or, if name is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen’s shape dictionary. Initially there are the following polygon shapes: “arrow”, “turtle”, “circle”, “square”, “triangle”, “classic”. To learn about how to deal with shapes see Screen method *register_shape()*.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

매개변수 **rmode** – one of the strings “auto”, “user”, “noresize”

Set resizemode to one of the values: “auto”, “user”, “noresize”. If *rmode* is not given, return current resizemode. Different resizemodes have the following effects:

- “auto”: adapts the appearance of the turtle corresponding to the value of pensize.
- “user”: adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by *shapesize()*.
- “noresize”: no adaption of the turtle’s appearance takes place.

resizemode(“user”) is called by *shapesize()* when used with arguments.

```
>>> 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** – positive number
- **stretch_len** – positive number
- **outline** – positive number

Return or set the pen’s attributes x/y-stretchfactors and/or outline. Set resizemode to “user”. If and only if resizemode is set to “user”, the turtle will be displayed stretched according to its stretchfactors: *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shapes’s outline.

```
>>> 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** – number (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do *not* change the turtle’s heading (direction of movement). If shear is not given: return the current shearfactor, i. e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

매개변수 **angle** – a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(*angle*)

매개변수 **angle** – a number

Rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

버전 3.1부터 폐지.

turtle.**tiltangle**(*angle=None*)

매개변수 **angle** – a number (optional)

Set or return the current tilt-angle. If angle is given, rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do *not* change the turtle's heading (direction of movement). If angle is not given: return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> 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** – a number (optional)
- **t12** – a number (optional)

- **t21** – a number (optional)
- **t12** – a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, 22. The determinant t11 * t22 - t12 * t21 must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> 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**()

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Using events

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

매개변수

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is `None`, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

매개변수

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is `None`, existing bindings are removed.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow)  # releasing turns it to transparent.
```

`turtle.`**`ondrag`**(*fun*, *btn=1*, *add=None*)

매개변수

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is `None`, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

Special Turtle methods

`turtle.`**`begin_poly`**()
Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

`turtle.`**`end_poly`**()
Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

`turtle.`**`get_poly`**()
Return the last recorded polygon.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

`turtle.`**`clone`**()
Create and return a clone of the turtle with same position, heading and turtle properties.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()
turtle.**getpen**()
Return the Turtle object itself. Only reasonable use: as a function to return the "anonymous turtle":

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()
Return the *TurtleScreen* object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

매개변수 **size** – an integer or None

Set or disable undobuffer. If *size* is an integer an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the *undo()* method/function. If *size* is None, the undobuffer is disabled.

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()
Return number of entries in the undobuffer.

```
>>> while undobufferentries():
...     undo()
```

Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class *Shape* explicitly as described below:

1. Create an empty Shape object of type "compound".
2. Add as many components to this object as desired, using the addcomponent() method.

   For example:

```
>>> 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. Now add the Shape to the Screen's shapelist and use it:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

참고: The *Shape* class is used internally by the *register_shape()* method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!

25.1.4 Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

Window control

`turtle.`**`bgcolor`**(**args*)

매개변수 **args** – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```
>>> 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 a gif-file or `"nopic"`, or `None`

Set background image or return name of current backgroundimage. If *picname* is a filename, set the corresponding image as background. If *picname* is `"nopic"`, delete background image, if present. If *picname* is `None`, return the filename of the current backgroundimage.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**()
`turtle.`**`clearscreen`**()
Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

참고: This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

`turtle.`**`reset`**()
`turtle.`**`resetscreen`**()
Reset all Turtles on the Screen to their initial state.

참고: This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

`turtle.`**`screensize`**(*canvwidth=None*, *canvheight=None*, *bg=None*)

매개변수

- **`canvwidth`** – positive integer, new width of canvas in pixels
- **`canvheight`** – positive integer, new height of canvas in pixels
- **`bg`** – colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle ;-)

`turtle.`**`setworldcoordinates`**(*llx*, *lly*, *urx*, *ury*)

매개변수

- **`llx`** – a number, x-coordinate of lower left corner of canvas
- **`lly`** – a number, y-coordinate of lower left corner of canvas
- **`urx`** – a number, x-coordinate of upper right corner of canvas
- **`ury`** – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode “world” if necessary. This performs a `screen.reset()`. If mode “world” is already active, all drawings are redrawn according to the new coordinates.

ATTENTION: in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

Animation control

turtle.**delay**(*delay=None*)

매개변수 **delay** – positive integer

Set or return the drawing *delay* in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

매개변수

- **n** – nonnegative integer
- **delay** – nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of n. Second argument sets delay value (see *delay()*).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()
Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method *speed()*.

Using screen events

turtle.**listen**(*xdummy=None*, *ydummy=None*)
Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass *listen()* to the onclick method.

turtle.**onkey**(*fun*, *key*)
turtle.**onkeyrelease**(*fun*, *key*)

매개변수

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind *fun* to key-release event of key. If *fun* is `None`, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

매개변수

- **fun** – a function with no arguments or None
- **key** – a string: key (e.g. "a") or key-symbol (e.g. "space")

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method *listen()*.)

```
>>> 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** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this screen. If *fun* is None, existing bindings are removed.

Example for a TurtleScreen instance named screen and a Turtle instance named turtle:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

참고: This TurtleScreen method is available as a global function only under the name onscreenclick. The global function onclick is another one derived from the Turtle method onclick.

turtle.**ontimer**(*fun*, *t=0*)

매개변수

- **fun** – a function with no arguments
- **t** – a number >= 0

Install a timer that calls *fun* after *t* milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**()
turtle.**done**()

Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

Input methods

turtle.**textinput**(*title*, *prompt*)

매개변수

- **title** – string
- **prompt** – string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

매개변수

- **title** – string
- **prompt** – string
- **default** – number (optional)
- **minval** – number (optional)
- **maxval** – number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Settings and special methods

turtle.**mode**(*mode=None*)

매개변수 **mode** – one of the strings “standard”, “logo” or “world”

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old *turtle*. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. **Attention**: in this mode angles appear distorted if `x/y` unit-ratio doesn’t equal 1.

Mode	Initial turtle heading	positive angles
“standard”	to the right (east)	counterclockwise
“logo”	upward (north)	clockwise

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

매개변수 **cmode** – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently *r*, *g*, *b* values of color triples have to be in the range 0..*cmode*.

```
>>> 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**()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

Return a list of names of all currently available turtle shapes.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)
turtle.**addshape**(*name*, *shape=None*)

There are three different ways to call this function:

(1) *name* is the name of a gif-file and *shape* is `None`: Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

참고: Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

(2) *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* is an arbitrary string and shape is a (compound) *Shape* object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

`turtle.`**`turtles`**`()`

Return the list of turtles on the screen.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`

Return the height of the turtle window.

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`

Return the width of the turtle window.

```
>>> screen.window_width()
640
```

Methods specific to Screen, not inherited from TurtleScreen

`turtle.`**`bye`**`()`

Shut the turtlegraphics window.

`turtle.`**`exitonclick`**`()`

Bind bye() method to mouse clicks on the Screen.

If the value "using_IDLE" in the configuration dictionary is `False` (default value), also enter mainloop. Remark: If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE's own mainloop is active also for the client script.

`turtle.`**`setup`**`(`*width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"]*`)`

Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

매개변수

- **width** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen

- **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- **startx** – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if None, center window horizontally
- **starty** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if None, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

turtle.**title**(*titlestring*)

매개변수 **titlestring** – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

25.1.5 Public classes

class turtle.**RawTurtle**(*canvas*)
class turtle.**RawPen**(*canvas*)

매개변수 **canvas** – a tkinter.Canvas, a *ScrolledCanvas* or a *TurtleScreen*

Create a turtle. The turtle has all methods described above as "methods of Turtle/RawTurtle".

class turtle.**Turtle**
Subclass of RawTurtle, has the same interface but draws on a default *Screen* object created automatically when needed for the first time.

class turtle.**TurtleScreen**(*cv*)

매개변수 **cv** – a tkinter.Canvas

Provides screen oriented methods like setbg() etc. that are described above.

class turtle.**Screen**
Subclass of TurtleScreen, with *four methods added*.

class turtle.**ScrolledCanvas**(*master*)

매개변수 **master** – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

class turtle.**Shape**(*type_*, *data*)

매개변수 **type_** – one of the strings "polygon", "image", "compound"

Data structure modeling shapes. The pair (type_, data) must follow this specification:

*type_*	*data*
"polygon"	a polygon-tuple, i.e. a tuple of pairs of coordinates
"image"	an image (in this form only used internally!)
"compound"	None (a compound shape has to be constructed using the *addcomponent()* method)

addcomponent(*poly*, *fill*, *outline=None*)

매개변수

- **poly** – a polygon, i.e. a tuple of pairs of numbers
- **fill** – a color the *poly* will be filled with
- **outline** – a color for the poly's outline (if given)

Example:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See *Compound shapes*.

class turtle.**Vec2D**(*x*, *y*)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for *a*, *b* vectors, *k* number):

- `a + b` vector addition
- `a - b` vector subtraction
- `a * b` inner product
- `k * a` and `a * k` multiplication with scalar
- `abs(a)` absolute value of a
- `a.rotate(angle)` rotation

25.1.6 Help and configuration

How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling *help()* on methods or functions displays the docstrings:

```
>>> 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"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      >>> 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

    No argument

    >>> turtle.penup()
```

- The docstrings of the functions which are derived from methods have a modified form:

```
>>> 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()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

turtle.**write_docstringdict**(*filename="turtle_docstringdict"*)

매개변수 **filename** – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script *filename*.py. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use *turtle* with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. turtle_docstringdict_german.py.

If you have an appropriate entry in your turtle.cfg file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file turtle.cfg which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following turtle.cfg:

```
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
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the Screen.setup() method.
- Line 5 and 6 correspond to the arguments of the method Screen.screensize().
- *shape* can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try help(shape).

- If you want to use no fillcolor (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg-file).
- If you want to reflect the turtle its state, you have to use `resizemode = auto`.
- If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as *turtle*.
- The entries *exampleturtle* and *examplescreen* define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- *using_IDLE*: Set this to `True` if you regularly work with IDLE and its -n switch ("no subprocess"). This will prevent *exitonclick()* to enter the mainloop.

There can be a `turtle.cfg` file in the directory where *turtle* is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

25.1.7 `turtledemo` — Demo scripts

The *turtledemo* package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows:

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The *turtledemo* package directory contains:

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the *turtle* module. Examples can be accessed via the Examples menu. They can also be run standalone.
- A `turtle.cfg` file which serves as an example of how to write and use such files.

The demo scripts are:

Name	Description	Features
bytedesign	complex classical turtle graphics pattern	`tracer()`, delay, `update()`
chaos	graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations	world coordinates
clock	analog clock showing time of your computer	turtles as clock's hands, ontimer
colormixer	experiment with r, g, b	`ondrag()`
forest	3 breadth-first trees	randomization
fractalcurves	Hilbert & Koch curves	recursion
lindenmayer	ethnomathematics (indian kolams)	L-System
minimal_hanoi	Towers of Hanoi	Rectangular Turtles as Hanoi discs (shape, shapesize)
nim	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
paint	super minimalistic drawing program	`onclick()`
peace	elementary	turtle: appearance and animation
penrose	aperiodic tiling with kites and darts	`stamp()`
planet_and_moon	simulation of gravitational system	compound shapes, `Vec2D`
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, clone shapesize, tilt, get_shapepoly, update
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	a (graphical) breadth first tree (using generators)	`clone()`
two_canvases	simple design	turtles on two canvases
wikipedia	a pattern from the wikipedia article on turtle graphics	`clone()`, `undo()`
yinyang	another elementary example	`circle()`

Have fun!

25.1.8 Changes since Python 2.6

- The methods `Turtle.tracer()`, `Turtle.window_width()` and `Turtle.window_height()` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen`/`Screen`-methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly: now every filling-process must be completed with an `end_fill()` call.
- A method `Turtle.filling()` has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

25.1.9 Changes since Python 3.0

- The methods `Turtle.shearfactor()`, `Turtle.shapetransform()` and `Turtle.get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `Turtle.tiltangle()` has been enhanced in functionality: it now can be used to get or set the tiltangle. `Turtle.settiltangle()` has been deprecated.
- The method `Screen.onkeypress()` has been added as a complement to `Screen.onkey()` which in fact binds actions to the keyrelease event. Accordingly the latter has got an alias: `Screen.onkeyrelease()`.
- The method `Screen.mainloop()` has been added. So when working only with Screen and Turtle objects one must not additionally import `mainloop()` anymore.
- Two input methods has been added `Screen.textinput()` and `Screen.numinput()`. These popup input dialogs and return strings and numbers respectively.
- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.

25.2 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.

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.

25.2.1 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 foo를 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*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.**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.)

25.2.2 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 cmdqueue에 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # ----- 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):
        '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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            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
(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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
(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
```

25.3 shlex — Simple lexical analysis

Source code: Lib/shlex.py

The *shlex* class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The *shlex* module defines the following functions:

shlex.**split**(*s*, *comments=False*, *posix=True*)
Split the string *s* using shell-like syntax. If *comments* is *False* (the default), the parsing of comments in the given string will be disabled (setting the *commenters* attribute of the *shlex* instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

참고: Since the *split()* function instantiates a *shlex* instance, passing None for *s* will read the string to split from standard input.

---

shlex.**quote**(*s*)
Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

quote() lets you plug the security hole:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with *split()*:

```
>>> 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 ~']
```

버전 3.3에 추가.

The *shlex* module defines the following class:

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A *shlex* instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with *read()* and *readline()* methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the *infile* attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument defines the operational mode: when *posix* is not true (default), the *shlex* instance will operate in compatibility mode. When operating in POSIX mode, *shlex* will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the *wordchars* attribute that appear in *punctuation_chars* will be removed from *wordchars*. See *Improved Compatibility with Shells* for more information. *punctuation_chars* can be set only upon *shlex* instance creation and can't be modified later.

버전 3.6에서 변경: The *punctuation_chars* parameter was added.

더 보기:

Module *configparser* Parser for configuration files similar to the Windows `.ini` files.

25.3.1 shlex Objects

A *shlex* instance has the following methods:

shlex.**get_token**()

Return a token. If tokens have been stacked using *push_token()*, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, *eof* is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

shlex.**push_token**(*str*)

Push the argument onto the token stack.

shlex.**read_token**()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.**sourcehook**(*filename*)
When *shlex* detects a source request (see *source* below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with *open()* called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

shlex.**push_source**(*newstream*, *newfile=None*)
Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

shlex.**pop_source**()
Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.**error_leader**(*infile=None*, *lineno=None*)
This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.**commenters**
The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.**wordchars**
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there.

shlex.**whitespace**
Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.**escape**
Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.**quotes**
Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.**escapedquotes**
Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'"'` by default.

shlex.**whitespace_split**
If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. If this attribute is `True`, *punctuation_chars* will have no effect, and splitting will happen only on whitespaces. When using *punctuation_chars*, which is intended to provide parsing closer to that implemented by shells, it is advisable to leave `whitespace_split` as `False` (the default value).

shlex.**infile**
The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.**instream**
The input stream from which this *shlex* instance is reading characters.

shlex.**source**
This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.**debug**
If this attribute is numeric and `1` or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.**lineno**
Source line number (count of newlines seen so far plus one).

shlex.**token**
The token buffer. It may be useful to examine this when catching exceptions.

shlex.**eof**
Token used to determine end of file. This will be set to the empty string (`''`), in non-POSIX mode, and to `None` in POSIX mode.

shlex.**punctuation_chars**
A read-only property. Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, '»>' could be returned as a token, even though it may not be recognised as such by shells.

버전 3.6에 추가.

25.3.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `"Do"` and `Separate`);

- If *whitespace_split* is `False`, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is `True`, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a *None* value;
- Quoted empty strings (`''`) are allowed.

25.3.3 Improved Compatibility with Shells

버전 3.6에 추가.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> list(shlex.shlex(text))
['a', '&', '&', 'b', ';', 'c', '&', '&', 'd', '|', '|', 'e', ';', 'f', '>',
"'abc'", ';', '(', 'def', '"ghi"', ')']
>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', "'abc'",
';', '(', 'def', '"ghi"', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

참고: When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)

CHAPTER 26

Tk를 사용한 그래픽 사용자 인터페이스

Tk/Tcl은 오랫동안 파이썬의 중요한 부분이었습니다. 견고하고 플랫폼 독립적인 윈도우 도구상자(파이썬 프로그래머는 *tkinter* 패키지를 통해 사용할 수 있습니다)와 그 확장(*tkinter.tix*와 *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. 또한, 내부 모듈 `_tkinter`는 파이썬과 Tcl이 상호 작용할 수 있게 해주는 스레드 안전한 메커니즘을 제공합니다.

*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. 대안에 관한 자세한 내용은 기타 그래픽 사용자 인터페이스 패키지 섹션을 참조하십시오.

26.1 `tkinter` — Tcl/Tk 파이썬 인터페이스

소스 코드: Lib/tkinter/__init__.py

---

tkinter 패키지("Tk 인터페이스")는 Tk GUI 툴킷에 대한 표준 파이썬 인터페이스입니다. Tk와 *tkinter*는 대부분의 유닉스 플랫폼과 윈도우 시스템에서 사용할 수 있습니다. (Tk 자체는 파이썬 일부가 아닙니다; ActiveState에서 유지 보수됩니다.)

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.

더 보기:

Tkinter 설명서:

파이썬 Tkinter 자원 파이썬 Tkinter 주제 지침서는 파이썬에서 Tk를 사용하는 것에 관한 많은 정보와 Tk에 관한 다른 정보 소스에 대한 링크를 제공합니다.

TKDocs 폭넓은 자습서와 일부 위젯(widget)에 대한 더 친절한 위젯 페이지.

Tkinter 8.5 reference: a GUI for Python 온라인 레퍼런스 자료.

effbot의 Tkinter 설명서 effbot.org에서 지원하는 tkinter의 온라인 레퍼런스.

Programming Python Mark Lutz의 책, 탁월하게 Tkinter를 다루고 있습니다.

Modern Tkinter for Busy Python Developers Book by Mark Roseman about building attractive and modern graphical user interfaces with Python and Tkinter.

Python and Tkinter Programming John Grayson의 책(ISBN 1-884777-81-3).

Tcl/Tk 설명서:

Tk commands 대부분의 명령은 *tkinter*나 *tkinter.ttk* 클래스로 사용할 수 있습니다. '8.6'을 여러분의 Tcl/Tk 설치 버전으로 바꾸십시오.

Tcl/Tk 최근 매뉴얼 페이지 www.tcl.tk에 있는 최근 Tcl/Tk 매뉴얼.

ActiveState Tcl 홈페이지 Tk/Tcl 개발은 주로 ActiveState에서 진행됩니다.

Tcl and the Tk Toolkit Tcl의 발명가인 John Ousterhout의 책.

Practical Programming in Tcl and Tk Brent Welch의 백과사전식 책.

26.1.1 Tkinter 모듈

대부분은, *tkinter* 만 있으면 충분하지만, 많은 추가 모듈도 사용할 수 있습니다. Tk 인터페이스는 `_tkinter` 라는 바이너리 모듈에 있습니다. 이 모듈은 Tk에 대한 저수준 인터페이스를 포함하고 있고, 응용 프로그램 프로그래머가 직접 사용해서는 안 됩니다. 일반적으로 공유 라이브러리(또는 DLL)이지만, 때에 따라 파이썬 인터프리터에 정적으로 링크될 수도 있습니다.

Tk 인터페이스 모듈 외에도, *tkinter*에는 파이썬 모듈이 많이 포함되어 있습니다. `tkinter.constants`는 가장 중요한 모듈 중 하나입니다. *tkinter*를 임포트 하면 `tkinter.constants`를 자동으로 임포트 하므로, 일반적으로 Tkinter를 사용하려면 간단한 import 문만 필요합니다:

```
import tkinter
```

또는, 더 자주:

```
from tkinter import *
```

class tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=1*)

Tk 클래스는 인자 없이 인스턴스화됩니다. 이것은 Tk의 최상위 위젯을 만드는데, 일반적으로 응용 프로그램의 메인 창입니다. 인스턴스마다 고유한 Tcl 인터프리터가 연결됩니다.

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=0*)

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 다른 모듈은 다음과 같습니다:

tkinter.scrolledtext 세로 스크롤 막대가 내장된 Text 위젯.

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.simpledialog 기본 대화 상자와 편리 함수.

tkinter.dnd *tkinter*를 위한 드래그 앤드 드롭 지원. 이것은 실험적이며 Tk DND로 대체 될 때 폐지됩니다.

turtle Tk 창에서의 터틀(turtle) 그래픽.

26.1.2 Tkinter 구명조끼

이 절은 Tk나 Tkinter에 대한 철저한 자습서가 되고자 하는 것은 아닙니다. 오히려, 시스템에 관한 몇 가지 입문 오리엔테이션을 제공하는 임시방편입니다.

크레딧:

- Tk는 John Ousterhout이 버클리에 있을 때 쓴 것입니다.
- Tkinter는 Steen Lumholt와 Guido van Rossum이 썼습니다.
- 이 구명조끼는 University of Virginia의 Matt Conway가 썼습니다.
- HTML 렌더링은, 그리고 일부 자유로운 편집과 함께, Ken Manheimer가 FrameMaker 버전으로 제작했습니다.
- Fredrik Lundh는 클래스 인터페이스 설명을 다듬고 수정하여 Tk 4.2에서 최신 버전으로 만들었습니다.
- Mike Clarkson은 설명서를 LaTeX로 변환하고, 레퍼런스 설명서의 사용자 인터페이스 장을 엮었습니다.

이 절을 사용하는 방법

이 절은 두 부분으로 구성되어 있습니다: 첫 번째 (대략) 절반은 배경 자료를 다루고, 두 번째 절반은 따다 쓰기에 간편한 레퍼런스입니다.

"어쩌고를 어떻게 해야 합니까?"와 같은 형식의 질문에 대답하려 할 때, Tk에서 직접 "어쩌고" 하는 법을 알아내고, 이것을 다시 해당 *tkinter* 호출로 변환하는 것이 종종 최선입니다. 파이썬 프로그래머는 Tk 설명서를 보고 종종 올바른 파이썬 명령을 추측할 수 있습니다. 이것은 Tkinter를 사용하려면 Tk에 대해 조금은 알고 있어야 한다는 뜻입니다. 이 문서가 그 소임을 수행하기는 부족하므로, 우리가 할 수 있는 최선은 존재하는 최고의 설명서를 소개하는 것입니다. 여기 몇 가지 힌트가 있습니다:

- 저자는 Tk 매뉴얼 페이지의 복사본을 얻을 것을 강력히 제안합니다. 특히, `manN` 디렉터리의 매뉴얼 페이지가 가장 유용합니다. `man3` 매뉴얼 페이지는 Tk 라이브러리에 대한 C 인터페이스를 설명하므로 스크립트 작성자에게 특별히 도움이 되지는 않습니다.
- Addison-Wesley는 초보자를 위한 Tcl과 Tk에 대한 훌륭한 소개인 John Ousterhout의 Tcl and the Tk Toolkit (ISBN 0-201-63337-X)이라는 책을 출간합니다. 이 책은 모든 것을 다루지는 않으며, 많은 세부 사항은 매뉴얼 페이지에 위임합니다.
- `tkinter/__init__.py`는 대부분에게 최후의 수단이지만, 다른 모든 것에서 답을 찾을 수 없을 때 가야 할 좋은 곳일 수 있습니다.

간단한 Hello World 프로그램

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.quit = tk.Button(self, text="QUIT", fg="red",
                              command=self.master.destroy)
        self.quit.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

26.1.3 Tcl/Tk (아주 빨리) 훑어보기

클래스 계층 구조가 복잡해 보입니다만, 실제로는 응용 프로그램 프로그래머는 거의 항상 계층 구조의 바닥에 있는 클래스를 참조합니다.

노트:

- 이 클래스들은 하나의 이름 공간에서 특정 기능을 구성하기 위해 제공됩니다. 이들은 독립적으로 인스턴스화하려는 것이 아닙니다.
- *Tk* 클래스는 응용 프로그램에서 한 번만 인스턴스화됩니다. 응용 프로그램 프로그래머는 명시적으로 인스턴스화 할 필요가 없으며, 다른 클래스 중 하나가 인스턴스화 될 때 시스템이 만듭니다.
- `Widget` 클래스는 인스턴스화하는 용도가 아니며, "실제" 위젯을 만들기 위해 서브 클래싱하려는 것입니다 (C++에서, 이것은 '추상 클래스(abstract class)'라고 합니다).

이 레퍼런스 자료를 사용하기 위해, Tk의 짧은 구문을 읽는 방법과 Tk 명령의 여러 부분을 식별하는 방법을 알아야 할 때가 있을 것입니다. (아래에 나오는 것의 *tkinter* 등가물에 대해서는 기본 *Tk*를 *Tkinter*로 매핑하기 절을 참조하십시오.)

Tk 스크립트는 Tcl 프로그램입니다. 모든 Tcl 프로그램과 마찬가지로, Tk 스크립트는 스페이스로 구분된 토큰 목록일 뿐입니다. Tk 위젯은 단지 그것의 클래스*(class)*, 그것을 구성하는 데 도움이 되는 옵션*(options)* 및 그것이 유용한 일을 하도록 하는 액션*(actions)*일 뿐입니다.

Tk에서 위젯을 만들려면, 명령은 항상 다음과 같은 형식입니다:

```
classCommand newPathname options
```

classCommand 어떤 종류의 위젯(버튼, 레이블, 메뉴…)을 만들지를 나타냅니다

newPathname 이 위젯의 새 이름입니다. Tk의 모든 이름은 고유해야 합니다. 이 기능을 강제하기 위해, Tk의 위젯은 파일 시스템의 파일과 마찬가지로, 경로명*(pathnames)*으로 이름이 지정됩니다. 최상위 수준 위젯, 루트*(root)*, 는 .(마침표)로 표현하고, 자식들은 더 많은 마침표로 구분합니다. 예를 들어, .myApp.controlPanel.okButton은 위젯의 이름일 수 있습니다.

options 위젯의 모양과 때에 따라 그 동작을 구성합니다. 옵션은 플래그와 값의 목록 형태로 제공됩니다. 플래그는 유닉스 셸 명령 플래그처럼 '-'가 앞에 오고, 값이 두 단어 이상이면 값을 따옴표로 묶습니다.

예를 들면:

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |                |
 class    new             options
command  widget  (-opt val -opt val ...)
```

일단 만들어지면, 위젯에 대한 경로명이 새로운 명령이 됩니다. 이 새로운 위젯 명령*(widget command)*은 액션*(action)*을 수행할 새 위젯을 얻는 데 필요한 프로그래머의 손잡이입니다. C에서, 이것을 someAction(fred, someOptions)로 표현하고, C++에서는, fred.someAction(someOptions)으로 표현하며, Tk에서는, 다음과 같이 표현합니다:

```
.fred someAction someOptions
```

객체 이름 .fred가 점으로 시작함에 유의하십시오.

예상대로, *someAction*의 유효한 값은 위젯의 클래스에 따라 다릅니다: .fred disable은 fred가 버튼이면 작동하지만, fred가 레이블이면 작동하지 않습니다(레이블의 비활성화는 Tk에서 지원되지 않습니다).

*someOptions*의 합법적인 값은 액션에 따라 다릅니다. disable과 같은 일부 액션에는 인자가 필요하지 않으며, 텍스트 입력 상자의 delete 명령과 같은 다른 액션에는 삭제할 텍스트 범위를 지정하는 인자가 필요합니다.

26.1.4 기본 Tk를 Tkinter로 매핑하기

Tk의 클래스 명령은 Tkinter의 클래스 생성자에 대응합니다.

```
button .fred                =====>  fred = Button()
```

객체의 마스터(master)는 생성 시 부여된 새로운 이름에 함축되어 있습니다. Tkinter에서, 마스터는 명시적으로 지정됩니다.

```
button .panel.fred          =====>  fred = Button(panel)
```

Tk의 구성 옵션은 뒤에 값이 오는 하이픈이 붙은 태그의 목록입니다. Tkinter에서 옵션은 인스턴스 생성자에서 키워드 인자로, 구성(configure) 호출에서는 키워드 인자로, 설정된 인스턴스에서는 딕셔너리 스타일의 인스턴스 인덱스로 지정됩니다. 옵션 설정에 대해서는 옵션 설정 절을 참조하십시오.

```
button .fred -fg red        =====>  fred = Button(panel, fg="red")
.fred configure -fg red     =====>  fred["fg"] = red
                            OR ==>  fred.config(fg="red")
```

Tk에서, 위젯에 대한 작업을 수행하려면, 위젯 이름을 명령으로 사용하고, 그 뒤에 액션 이름이 옵니다, 인자(옵션)가 붙는 것도 가능합니다. Tkinter에서는, 위젯의 액션을 호출하기 위해 클래스 인스턴스의 메서드를 호출합니다. 주어진 위젯이 수행할 수 있는 액션(메서드)은 tkinter/__init__.py에 나열됩니다.

```
.fred invoke                =====>  fred.invoke()
```

패커(packer)(지오메트리 관리자, geometry manager)에게 위젯을 전달하려면, pack을 선택적 인자로 호출합니다. Tkinter에서, Pack 클래스는 이 모든 기능을 담고 있으며, 다양한 형태의 pack 명령이 메서드로 구현됩니다. *tkinter*의 모든 위젯은 Pack의 서브 클래스이므로, 모든 패킹 메서드를 상속받습니다. Form 지오메트리 관리자에 대한 추가 정보는 *tkinter.tix* 모듈 설명서를 참조하십시오.

```
pack .fred -side left       =====>  fred.pack(side="left")
```

26.1.5 Tk와 Tkinter의 관계

위에서 아래로:

여러분의 응용 프로그램이 여기에 있습니다 (파이썬) 파이썬 응용 프로그램이 *tkinter* 호출을 합니다.

tkinter (파이썬 패키지) 이 호출(예를 들어, 버튼 위젯을 만든다고 합시다)은 파이썬으로 작성된 *tkinter* 패키지에 구현되어 있습니다. 이 파이썬 함수는 명령과 인자를 구문 분석하여, 파이썬 스크립트 대신 Tk 스크립트에서 나온 것처럼 보이는 형식으로 변환합니다.

_tkinter (C) 이 명령과 인자는 `_tkinter` - 밑줄에 주목하세요 - 확장 모듈의 C 함수에 전달됩니다.

Tk 위젯 (C와 Tcl) 이 C 함수는 Tk 라이브러리를 구성하는 C 함수를 포함하는 다른 C 모듈을 호출할 수 있습니다. Tk는 C와 약간의 Tcl로 구현됩니다. Tk 위젯의 Tcl 부분은 어떤 기본 동작을 위젯에 연결하는 데 사용되며, 파이썬 *tkinter* 패키지가 임포트 되는 시점에 한 번 실행됩니다. (사용자는 이 단계를 결코 보지 못합니다).

Tk (C) Tk 위젯의 Tk 부분은 최종 … 로의 매핑을 구현합니다

Xlib (C) 화면에 그래픽을 그리기 위한 Xlib 라이브러리.

26.1.6 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 반환 값은 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 위젯 위치의 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()                     # defaults to 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 183쪽을 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:

```
class App(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("hi. contents of entry is now ---->",
              self.contents.get())
```

창 관리자

Tk에는, 창 관리자와 상호 작용하기 위한 유틸리티 명령인 wm이 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()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
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.h의 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: c는 센티미터, i는 인치, m은 밀리미터, p는 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 메서드도 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 매뉴얼 페이지와 John Ousterhout의 책의 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).

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.

26.1.7 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.

26.2 `tkinter.ttk` — Tk themed widgets

Source code: Lib/tkinter/ttk.py

The *tkinter.ttk* module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if *Tile* has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for *tkinter.ttk* is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

더 보기:

Tk Widget Styling Support A document introducing theming support for Tk

26.2.1 Using Ttk

To start using Ttk, import its module:

```
from tkinter import ttk
```

To override the basic Tk widgets, the import should follow the Tk import:

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several *tkinter.ttk* widgets (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

더 보기:

Converting existing applications to use Tile widgets A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

26.2.2 Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar`, and *Spinbox*. The other six are new: *Combobox*, *Notebook*, *Progressbar*, `Separator`, `Sizegrip` and *Treeview*. And all them are subclasses of *Widget*.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code:

```
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")
```

For more information about *TtkStyling*, see the *Style* class documentation.

26.2.3 Widget

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the `ttk` Widgets accepts the following options:

Option	Description
class	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This option is read-only, and may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.
style	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

Option	Description
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget's window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method `Scrollbar.set()` of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For some more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

Option	Description
text	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by `Style.map()`, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Compatibility Options

Option	Description
state	May be set to "normal" or "disabled" to control the "disabled" state bit. This is a write-only option: setting it changes the widget state, but the `Widget.state()` method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

Flag	Description
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur
disabled	Widget is disabled under program control
focus	Widget has keyboard focus
pressed	Widget is being pressed
selected	"On", "true", or "current" for things like Checkbuttons and radiobuttons
background	Windows and Mac have a notion of an "active" or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window
readonly	Widget should not allow user modification
alternate	A widget-specific alternate display format
invalid	The widget's value is invalid

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is

off.

ttk.Widget

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`.

class tkinter.ttk.**Widget**

identify(*x*, *y*)
Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)
Test the widget's state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with args if widget state matches *statespec*.

state(*statespec=None*)
Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.

26.2.4 Combobox

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from *Widget*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

Options

This widget accepts the following specific options:

Option	Description
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).
justify	Specifies how the text is aligned within the widget. One of "left", "center", or "right".
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.
state	One of "normal", "readonly", or "disabled". In the "readonly" state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the "normal" state, the text field is directly editable. In the "disabled" state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `tkinter.StringVar`.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Virtual events

The combobox widgets generates a **«ComboboxSelected»** virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)
If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()
Returns the current value of the combobox.

set(*value*)
Sets the value of the combobox to *value*.

26.2.5 Spinbox

The `ttk.Spinbox` widget is a `ttk.Entry` enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of `Entry`.

Besides the methods inherited from *`Widget`*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, it has some other methods, described at `ttk.Spinbox`.

Options

This widget accepts the following specific options:

Option	Description
from	Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as `from_` when used as an argument, since `from` is a Python keyword.
to	Float value. If set, this is the maximum value to which the increment button will increment.
increment	Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.
values	Sequence of string or float values. If specified, the increment/decrement buttons will cycle through the items in this sequence rather than incrementing or decrementing numbers.
wrap	Boolean value. If `True`, increment and decrement buttons will cycle from the `to` value to the `from` value or the `from` value to the `to` value, respectively.
format	String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form "%W.Pf", where W is the padded width of the value, P is the precision, and '%' and 'f' are literal.
command	Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.

Virtual events

The spinbox widget generates an **«Increment»** virtual event when the user presses <Up>, and a **«Decrement»** virtual event when the user presses <Down>.

ttk.Spinbox

class tkinter.ttk.**Spinbox**

get()
Returns the current value of the spinbox.

set(*value*)
Sets the value of the spinbox to *value*.

26.2.6 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

Options

This widget accepts the following specific options:

Option	Description
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

Option	Description
state	Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in *`Widget`*.
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See *Label Options* for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if *`Notebook.enable_traversal()`* is called.

Tab Identifiers

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form “@x,y”, which identifies the tab
- The literal string “current”, which identifies the currently-selected tab
- The literal string “end”, which returns the number of tabs (only valid for *`Notebook.index()`*)

Virtual Events

This widget generates a **«NotebookTabChanged»** virtual event after a new tab is selected.

ttk.Notebook

class tkinter.ttk.**Notebook**

add(*child*, ***kw*)
Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See *Tab Options* for the list of available options.

forget(*tab_id*)
Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)
Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

identify(*x*, *y*)
Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index(*tab_id*)
Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string "end".

insert(*pos*, *child*, ***kw*)
Inserts a pane at the specified position.

pos is either the string "end", an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See *Tab Options* for the list of available options.

select(*tab_id=None*)
Selects the specified *tab_id*.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)
Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs()
Returns a list of windows managed by the notebook.

enable_traversal()
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.
- `Shift-Control-Tab`: selects the tab preceding the currently selected one.

- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

26.2.7 Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Options

This widget accepts the following specific options:

Option	Description
orient	One of "horizontal" or "vertical". Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	One of "determinate" or "indeterminate".
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In "determinate" mode, this represents the amount of work completed. In "indeterminate" mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one "cycle" when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)
Begin autoincrement mode: schedules a recurring timer event that calls *Progressbar.step()* every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step(*amount=None*)
Increments the progress bar's value by *amount*.

amount defaults to 1.0 if omitted.

stop()
Stop autoincrement mode: cancels any recurring timer event initiated by *Progressbar.start()* for this progress bar.

26.2.8 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

Options

This widget accepts the following specific option:

Option	Description
orient	One of "horizontal" or "vertical". Specifies the orientation of the separator.

26.2.9 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On MacOS X, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. `…`), the `Sizegrip` widget will not resize the window.
- This widget supports only "southeast" resizing.

26.2.10 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods *`Treeview.xview()`* and *`Treeview.yview()`*.

Options

This widget accepts the following specific options:

Option	Description
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all".
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of "extended", "browse" or "none". If set to "extended" (the default), multiple items may be selected. If "browse", only a single item will be selected at a time. If "none", the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is "tree headings", i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show="tree" is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

Option	Description
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	`True`/`False` value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

Option	Description
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item's image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notes:

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

Event	Description
«TreeviewSelect»	Generated whenever the selection changes.
«TreeviewOpen»	Generated just before settings the focus item to open=True.
«TreeviewClose»	Generated just after setting the focus item to open=False.

The `Treeview.focus()` and `Treeview.selection()` methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)
Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*item=None*)
Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)
Replaces *item*'s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*'s children.

column(*column*, *option=None*, ***kw*)
Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **id** Returns the column name. This is a read-only option.
- **anchor: One of the standard Tk anchor values.** Specifies how the text in this column should be aligned with respect to the cell.
- **minwidth: width** The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- **stretch: `True/False`** Specifies whether the column's width should be adjusted when the widget is resized.
- **width: width** The width of the column in pixels.

To configure the tree column, call this with column = "#0"

delete(**items*)
Delete all specified *items* and all their descendants.

The root item may not be deleted.

detach(**items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists(*item*)
Returns `True` if the specified *item* is present in the tree.

focus(*item=None*)
If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or '' if there is none.

heading(*column*, *option=None*, ***kw*)
Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **text: text** The text to display in the column heading.
- **image: imageName** Specifies an image to display to the right of the column heading.

- **anchor: anchor** Specifies how the heading text should be aligned. One of the standard Tk anchor values.
- **command: callback** A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

identify (*component*, *x*, *y*)
Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

identify_row (*y*)
Returns the item ID of the item at position *y*.

identify_column (*x*)
Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region (*x*, *y*)
Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element (*x*, *y*)
Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index (*item*)
Returns the integer index of *item* within its parent's list of children.

insert (*parent*, *index*, *iid=None*, ***kw*)
Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See *Item Options* for the list of available points.

item (*item*, *option=None*, ***kw*)
Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move (*item*, *parent*, *index*)
Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next (*item*)
Returns the identifier of *item*'s next sibling, or '' if *item* is the last child of its parent.

parent (*item*)
Returns the ID of the parent of *item*, or '' if *item* is at the top level of the hierarchy.

prev (*item*)
Returns the identifier of *item*'s previous sibling, or '' if *item* is the first child of its parent.

reattach (*item*, *parent*, *index*)
An alias for `Treeview.move()`.

see (*item*)
Ensure that *item* is visible.

Sets all of *item*'s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection (*selop=None*, *items=None*)
If *selop* is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

Deprecated since version 3.6, will be removed in version 3.8: Using `selection()` for changing the selection state is deprecated. Use the following selection methods instead.

selection_set (**items*)
items becomes the new selection.

버전 3.6에서 변경: *items* can be passed as separate arguments, not just as a single tuple.

selection_add (**items*)
Add *items* to the selection.

버전 3.6에서 변경: *items* can be passed as separate arguments, not just as a single tuple.

selection_remove (**items*)
Remove *items* from the selection.

버전 3.6에서 변경: *items* can be passed as separate arguments, not just as a single tuple.

selection_toggle (**items*)
Toggle the selection state of each item in *items*.

버전 3.6에서 변경: *items* can be passed as separate arguments, not just as a single tuple.

set (*item*, *column=None*, *value=None*)
With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind (*tagname*, *sequence=None*, *callback=None*)
Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure (*tagname*, *option=None*, ***kw*)
Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has (*tagname*, *item=None*)
If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview (**args*)
Query or modify horizontal position of the treeview.

yview (**args*)
Query or modify vertical position of the treeview.

26.2.11 Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

더 보기:

Tcl'2004 conference presentation This document explains how the theme engine works

class tkinter.ttk.**Style**
This class is used to manipulate the style database.

configure (*style*, *query_opt=None*, ***kw*)
Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```
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*)
Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup (*style*, *option*, *state=None*, *default=None*)
Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style*, *layoutspec=None*)
Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in *Layouts*.

To understand the format, see the following example (it is not intended to do anything useful):

```
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*)
Create a new element in the current theme, of the given *etype* which is expected to be either “image”, “from” or “vsapi”. The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If “image” is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

- **border=padding** padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.
- **height=height** Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.
- **padding=padding** Specifies the element's interior padding. Defaults to border's value if not specified.
- **sticky=spec** Specifies how the image is placed within the final parcel. spec contains zero or more characters "n", "s", "w", or "e".
- **width=width** Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If "from" is used as the value of *etype*, `element_create()` will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

element_names()
Returns the list of elements defined in the current theme.

element_options(*elementname*)
Returns the list of *elementname*'s options.

theme_create(*themename*, *parent=None*, *settings=None*)
Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for `theme_settings()`.

theme_settings(*themename*, *settings*)
Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods `Style.configure()`, `Style.map()`, `Style.layout()` and `Style.element_create()` respectively.

As an example, let's change the Combobox for the default theme a bit:

```
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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
root.mainloop()
```

theme_names()
Returns a list of all known themes.

theme_use(*themename=None*)
If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a «ThemeChanged» event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- **side: whichside** Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- **sticky: nswe** Specifies where the element is placed inside its allocated parcel.
- **unit: 0 or 1** If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of *`Widget.identify()`* et al. It's used for things like scrollbar thumbs with grips.
- **children: [sublayout…]** Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a *Layout*.

26.3 `tkinter.tix` — Extension widgets for Tk

Source code: Lib/tkinter/tix.py

버전 3.6부터 폐지: This Tk extension is unmaintained and should not be used in new code. Use *`tkinter.ttk`* instead.

The *`tkinter.tix`* (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The *`tkinter.tix`* library provides most of the commonly needed widgets that are missing from standard Tk: *`HList`*, *`ComboBox`*, *`Control`* (a.k.a. SpinBox) and an assortment of scrollable widgets. *`tkinter.tix`* also includes many more widgets that are generally useful in a wide range of applications: *`NoteBook`*, *`FileEntry`*, *`PanedWindow`*, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

더 보기:

Tix Homepage The home page for `Tix`. This includes links to additional documentation and downloads.

Tix Man Pages On-line version of the man pages and reference material.

Tix Programming Guide On-line version of the programmer's reference material.

Tix Development Applications Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **`TixInspect`**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.

26.3.1 Using Tix

class tkinter.tix.**Tk**(*screenName=None*, *baseName=None*, *className='Tix'*)
Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the *tkinter.tix* module subclasses the classes in the *tkinter*. The former imports the latter, so to use *tkinter.tix* with Tkinter, all you need to do is to import one module. In general, you can just import *tkinter.tix*, and replace the toplevel call to *tkinter.Tk* with `tix.Tk`:

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

To use *tkinter.tix*, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

26.3.2 Tix Widgets

Tix introduces over 40 widget classes to the *tkinter* repertoire.

Basic Widgets

class tkinter.tix.**Balloon**
A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.**ButtonBox**
The ButtonBox widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

class tkinter.tix.**ComboBox**
The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.**Control**
The Control widget is also known as the `SpinBox` widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class tkinter.tix.**LabelEntry**
The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of "entry-form" type of interface.

class tkinter.tix.**LabelFrame**
The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

class tkinter.tix.**Meter**
The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class tkinter.tix.**OptionMenu**
The OptionMenu creates a menu button of options.

class tkinter.tix.**PopupMenu**
The PopupMenu widget can be used as a replacement of the `tk_popup` command. The advantage of the `Tix` *PopupMenu* widget is it requires less application code to manipulate.

class tkinter.tix.**Select**
The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class tkinter.tix.**StdButtonBox**
The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

File Selectors

class tkinter.tix.**DirList**
The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirTree**
The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirSelectDialog**
The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class tkinter.tix.**DirSelectBox**
The *DirSelectBox* is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class tkinter.tix.**ExFileSelectBox**
The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the *ExFileSelectBox* widget is very similar to the standard file dialog on MS Windows 3.1.

class tkinter.tix.**FileSelectBox**
The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a *ComboBox* widget so that they can be quickly selected again.

class tkinter.tix.**FileEntry**
The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class tkinter.tix.**HList**
The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class tkinter.tix.**CheckList**
The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk check-button or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class tkinter.tix.**Tree**
The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class tkinter.tix.**TList**
The TList widget can be used to display data in a tabular format. The list entries of a *TList* widget are similar to the entries in the Tk listbox widget. The main differences are (1) the *TList* widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Manager Widgets

class tkinter.tix.**PanedWindow**
The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class tkinter.tix.**ListNoteBook**
The ListNoteBook widget is very similar to the `TixNoteBook` widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

class tkinter.tix.**NoteBook**
The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual "tabs" at the top of the NoteBook widget.

Image Types

The *tkinter.tix* module adds:

- pixmap capabilities to all *tkinter.tix* and *tkinter* widgets to create color images from XPM files.
- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk `Button` widget.

Miscellaneous Widgets

class tkinter.tix.**InputOnly**
The InputOnly widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

Form Geometry Manager

In addition, *tkinter.tix* augments *tkinter* by providing:

class tkinter.tix.**Form**
The Form geometry manager based on attachment rules for all Tk widgets.

26.3.3 Tix Commands

class tkinter.tix.**tixCommand**

The tix commands provide access to miscellaneous elements of `Tix`'s internal state and the `Tix` application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.**tix_configure**(*cnf=None*, ***kw*)

Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.**tix_cget**(*option*)

Returns the current value of the configuration option given by *option*. Option may be any of the configuration options.

tixCommand.**tix_getbitmap**(*name*)

Locates a bitmap file of the name `name.xpm` or `name` in one of the bitmap directories (see the *tix_addbitmapdir()* method). By using *tix_getbitmap()*, you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character `@`. The returned value can be used to configure the `bitmap` option of the Tk and Tix widgets.

tixCommand.**tix_addbitmapdir**(*directory*)

Tix maintains a list of directories under which the *tix_getimage()* and *tix_getbitmap()* methods will search for image files. The standard bitmap directory is `$TIX_LIBRARY/bitmaps`. The *tix_addbitmapdir()* method adds *directory* into this list. By using this method, the image files of an applications can also be located using the *tix_getimage()* or *tix_getbitmap()* method.

tixCommand.**tix_filedialog**([*dlgclass*])

Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to *tix_filedialog()*. An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are `tix`, `FileSelectDialog` or `tixExFileSelectDialog`.

tixCommand.**tix_getimage**(*self*, *name*)

Locates an image file of the name `name.xpm`, `name.xbm` or `name.ppm` in one of the bitmap directories (see the *tix_addbitmapdir()* method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using *tix_getimage()*, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

tixCommand.**tix_option_get**(*name*)

Gets the options maintained by the Tix scheme mechanism.

tixCommand.**tix_resetoptions**(*newScheme*, *newFontSet*[, *newScmPrio*])

Resets the scheme and fontset of the Tix application to *newScheme* and *newFontSet*, respectively. This affects only

those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter *newScmPrio* can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and inited, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the *`tix_resetoptions()`* method must be used.

26.4 `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. 생성자는 `tkinter.Text` 클래스의 생성자와 같습니다.

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:

`ScrolledText.`**`frame`**
텍스트 및 스크롤 막대 위젯을 둘러싼 프레임.

`ScrolledText.`**`vbar`**
스크롤 막대 위젯.

26.5 IDLE

Source code: Lib/idlelib/

IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the *`tkinter`* GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

26.5.1 Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

File menu (Shell and Editor)

New File Create a new file editing window.

Open… Open an existing file with an Open dialog.

Recent Files Open a list of recent files. Click one to open it.

Open Module… Open an existing module (searches sys.path).

Class Browser Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As… Save the current window with a Save As dialog. The file saved becomes the new associated file for the window.

Save Copy As… Save the current window to different file without changing the associated file.

Print Window Print the current window to the default printer.

Close Close the current window (ask to save if unsaved).

Exit Close all windows and quit IDLE (ask to save unsaved windows).

Edit menu (Shell and Editor)

Undo Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo Redo the last undone change to the current window.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Select All Select the entire contents of the current window.

Find… Open a search dialog with many options

Find Again Repeat the last search, if there is one.

Find Selection Search for the currently selected string, if there is one.

Find in Files… Open a file search dialog. Put results in a new output window.

Replace⋯ Open a search-and-replace dialog.

Go to Line Move the cursor to the beginning of the line requested and make that line visible. A request past the end of the file goes to the end. Clear any selection and update the line and column status.

Show Completions Open a scrollable list allowing selection of keywords and attributes. See *Completions* in the Editing and navigation section below.

Expand Word Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show call tip After an unclosed parenthesis for a function, open a small window with function parameter hints. See *Calltips* in the Editing and navigation section below.

Show surrounding parens Highlight the surrounding parenthesis.

Format menu (Editor window only)

Indent Region Shift selected lines right by the indent width (default 4 spaces).

Dedent Region Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region Insert ## in front of selected lines.

Uncomment Region Remove leading # or ## from selected lines.

Tabify Region Turn *leading* stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region Turn *all* tabs into the correct number of spaces.

Toggle Tabs Open a dialog to switch between indenting with spaces and tabs.

New Indent Width Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Format Paragraph Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Strip trailing whitespace Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings. Except for Shell windows, remove extra newlines at the end of the file.

Run menu (Editor window only)

Run Module Do *Check Module*. If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Run⋯ Customized Same as *Run Module*, but run the module with customized settings. *Command Line Arguments* extend `sys.argv` as if passed on a command line. The module can be run in the Shell without restarting.

Check Module Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Python Shell Open or wake up the Python Shell window.

Shell menu (Shell window only)

View Last Restart Scroll the shell window to the last Shell restart.

Restart Shell Restart the shell to clean the environment.

Previous History Cycle through earlier commands in history which match the current entry.

Next History Cycle through later commands in history which match the current entry.

Interrupt Execution Stop a running program.

Debug menu (Shell window only)

Go to File/Line Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle) When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more details, see *Setting preferences* under Help and preferences.

Most configuration options apply to all windows or all future windows. The option items below only apply to the active window.

Show/Hide Code Context (Editor Window only) Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. See *Code Context* in the Editing and Navigation section below.

Show/Hide Line Numbers (Editor Window only) Open a column to the left of the edit window which shows the number of each line of text. The default is off, which may be changed in the preferences (see *Setting preferences*).

Zoom/Restore Height Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog. The maximum height for a screen is determined by momentarily maximizing a window the first time one is zoomed on the screen. Changing screen settings may invalidate the saved height. This toggle has no effect when a window is maximized.

Window menu (Shell and Editor)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE Display version, copyright, license, credits, and more.

IDLE Help Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Python Docs Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Turtle Demo Run the turtledemo module with example Python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the *Help sources* subsection below for more on Help menu choices.

Context Menus

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's `.idlerc` directory.

Set Breakpoint Set a breakpoint on the current line.

Clear Breakpoint Clear the breakpoint on that line.

Shell and Output windows also have the following.

Go to file/line Same as in Debug menu.

The Shell window also has an output squeezing facility explained in the *Python Shell window* subsection below.

Squeeze If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

26.5.2 Editing and navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number ('Ln') and column number ('Col'). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

In this section, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on macOS.

- `Backspace` deletes to the left; `Del` deletes to the right
- `C-Backspace` delete word left; `C-Del` delete word to the right
- Arrow keys and `Page Up`/`Page Down` to move around
- `C-LeftArrow` and `C-RightArrow` moves by words
- `Home`/`End` go to begin/end of line
- `C-Home`/`C-End` go to begin/end of file
- Some useful Emacs bindings are inherited from Tcl/Tk:
  - `C-a` beginning of line
  - `C-e` end of line
  - `C-k` kill line (but doesn't put it in clipboard)
  - `C-l` center window around the insertion point
  - `C-b` go backward one character without deleting (usually you can also use the cursor key for this)
  - `C-f` go forward one character without deleting (usually you can also use the cursor key for this)
  - `C-p` go up one line (usually you can also use the cursor key for this)
  - `C-d` delete next character

Standard keybindings (like `C-c` to copy and `C-v` to paste) may work. Keybindings are selected in the Configure IDLE dialog.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, `Backspace` deletes up to 4 spaces if they are there. `Tab` inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands on the *Format menu*.

Completions

Completions are supplied for functions, classes, and attributes of classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a '.' or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a `Tab` will supply that completion without opening the ACW.

'Show Completions' will force open a completions window, by default the `C-space` will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a `tab` will cause the longest non-ambiguous match to be entered in the Editor window or Shell. Two `tab` in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

"Hidden" attributes can be accessed by typing the beginning of hidden name after a '.', e.g. '_'. This allows access to modules with `__all__` set, or to class-private attributes.

Completions and the 'Expand Word' facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names in an Editor window which are not via `__main__` and *`sys.modules`* will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don't like the ACW popping up unbidden, simply make the delay longer or disable the extension.

Calltips

A calltip is shown when one types `(` after the name of an *accessible* function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip consists of all lines up the fifth line or the first blank line. These details may change.

The set of *accessible* functions depends on what modules have been imported into the user process, including those imported by Idle itself, and what definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not import turtle. The menu or shortcut do nothing either. Enter `import turtle` and then `turtle.write(` will work.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing the import statements at the top, or immediately run an existing file before editing.

Code Context

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

Python Shell window

With IDLE's Shell, one enters, edits, and recalls complete statements. Most consoles and terminals only work with a single physical line at a time.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`. One may edit pasted code first. If one pastes more that one statement into Shell, the result will be a *`SyntaxError`* when multiple statements are compiled as if they were one.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following keys.

- `C-c` interrupts executing command
- `C-d` sends end-of-file; closes window if typed at a >>> prompt
- `Alt-/` (Expand word) is also useful to reduce typing

  Command history

  - `Alt-p` retrieves previous command matching what you have typed. On macOS use `C-p`.
  - `Alt-n` retrieves next. On macOS use `C-n`.
  - `Return` while on any previous command retrieves that command

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

26.5.3 Startup and code execution

Upon startup with the `-s` option, IDLE will execute the file referenced by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

Command line usage

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

If there are arguments:

- If `-`, `-c`, or `r` is used, all arguments are placed in `sys.argv[1:...]` and `sys.argv[0]` is set to `''`, `'-c'`, or `'-r'`. No editor window is opened, even if that is the default set in the Options dialog.
- Otherwise, arguments are files opened for editing and `sys.argv` reflects the arguments passed to IDLE itself.

Startup failure

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to check for one and stop it if there is. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or using Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in `~/.idlerc/` (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand. Instead, use the configuration dialog, under Options. Once there is an error in a user configuration file, the best solution may be to delete it and start over with the settings dialog.

If IDLE quits with no message, and it was not started from a console, try starting it from a console or terminal (`python -m idlelib`) and see if this results in an error message.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.activeCount()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. The secondary subprocess will then be attached to that window for input and output.

The IDLE code running in the execution process adds frames to the call stack that would not be there otherwise. IDLE wraps `sys.getrecursionlimit` and `sys.setrecursionlimit` to reduce the effect of the additional stack frames.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE's changes are lost and input from the keyboard and output to the screen will not work correctly.

When user code raises SystemExit either directly or by calling sys.exit, IDLE returns to a Shell prompt instead of exiting.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

A Tk Text widget, and hence IDLE's Shell, displays characters (codepoints) in the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 'characters'). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

The `repr` function is used for interactive echo of expression values. It returns an altered version of the input string in which control codes, some BMP codepoints, and all non-BMP codepoints are replaced with escape codes. As demonstrated above, it allows one to identify the characters in a string, regardless of how they are displayed.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal '^' marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a 'Squeezed text' label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

버전 3.4부터 폐지.

26.5.4 Help and preferences

Help sources

Help menu entry “IDLE Help” displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry “Python Docs” opens the extensive sources of help, including tutorials, available at `docs.python.org/x.y`, where ‘x.y’ is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog.

Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a `.idlerc` directory in the user’s home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in `.idlerc`.

On the Font tab, see the text sample for the effect of font face and size on multiple characters in multiple languages. Edit the sample to add other characters of personal interest. Use the sample to select monospaced fonts. If particular characters have problems in Shell or an editor, add them to the top of the sample and try changing first size and then font.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

IDLE on macOS

Under System Preferences: Dock, one can set “Prefer tabs when opening documents” to “Always”. This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

26.6 기타 그래픽 사용자 인터페이스 패키지

주요 교차 플랫폼(윈도우, 맥 OS X, 유닉스 계열) GUI 도구상자를 파이썬에서 사용할 수 있습니다:

더 보기:

PyGObject PyGObject는 GObject를 사용하여 C 라이브러리에 대한 인트로스펙션 바인딩을 제공합니다. 이 라이브러리 중 하나가 GTK+ 3 위젯 집합입니다. GTK+에는 Tkinter가 제공하는 것보다 더 많은 위젯이 제공됩니다. 온라인 파이썬 GTK+ 3 자습서가 있습니다.

PyGTK PyGTK는 라이브러리의 이전 버전인 GTK+ 2에 대한 바인딩을 제공합니다. 이것은 C보다 약간 높은 수준의 객체 지향 인터페이스를 제공합니다. GNOME 바인딩도 있습니다. 온라인 자습서가 있습니다.

PyQt PyQt는 Qt 도구상자에 대한 **sip**-래핑 된 바인딩입니다. Qt는 유닉스, 윈도우 및 맥 OS X에서 사용할 수 있는 광범위한 C++ GUI 응용 프로그램 개발 프레임워크입니다. **sip**는 파이썬 클래스로 C++ 라이브러리에 대한 바인딩을 생성하는 도구이며, 파이썬 용으로 특별히 설계되었습니다.

PySide2 Also known as the Qt for Python project, PySide2 is a newer binding to the Qt toolkit. It is provided by The Qt Company and aims to provide a complete port of PySide to Qt 5. Compared to PyQt, its licensing scheme is friendlier to non-open source applications.

wxPython wxPython은 인기 있는 wxWidgets (이전 wxWindows) C++ 도구상자를 기반으로 작성된 파이썬 용 교차 플랫폼 GUI 도구상자입니다. 윈도우, 맥 OS X 및 유닉스 시스템의 응용 프로그램에 대해 고유한 모양과 느낌을 제공하는데, 가능한 각 플랫폼 고유의 위젯 집합(유닉스 계열 시스템에서는 GTK+)을 사용합니다. 광범위한 위젯 외에도 wxPython은 온라인 설명서 및 문맥에 맞는 도움말, 인쇄, HTML 보기, 저수준 장치 컨텍스트 그리기, 끌어서 놓기, 시스템 클립보드 액세스, XML 기반 자원 형식 등과 이밖에도 많은 것들을 위한 클래스를 제공하는데, 사용자 기여 모듈의 라이브러리는 계속 늘어나고 있습니다.

PyGTK, PyQt, PySide2, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the GUI Programming page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.

CHAPTER 27

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. **2to3**는 파이썬 2.x 소스 코드를 유효한 파이썬 3.x 코드로 변환할 수 있습니다.

이 장에서 설명하는 모듈 목록은 다음과 같습니다:

27.1 `typing` — Support for type hints

버전 3.5에 추가.

Source code: 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 supports type hints as specified by **PEP 484** and **PEP 526**. The most fundamental support consists of the types *Any*, *Union*, *Tuple*, *Callable*, *TypeVar*, and *Generic*. For full specification please see **PEP 484**. For a simplified introduction to type hints see **PEP 483**.

The function below takes and returns a string and is annotated as follows:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

In the function `greeting`, the argument `name` is expected to be of type *str* and the return type *str*. Subtypes are accepted as arguments.

27.1.1 Type aliases

A type alias is defined by assigning the type to the alias. In this example, `Vector` and `List[float]` will be treated as interchangeable synonyms:

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Type aliases are useful for simplifying complex type signatures. For example:

```
from typing import Dict, Tuple, Sequence

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
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:
    ...
```

Note that `None` as a type hint is a special case and is replaced by `type(None)`.

27.1.2 NewType

Use the *`NewType()`* helper function to create distinct types:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. This is useful in helping catch logical errors:

```
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

You may still perform all `int` operations on a variable of type `UserId`, but the result will always be of type `int`. This lets you pass in a `UserId` wherever an `int` might be expected, but will prevent you from accidentally creating a

`UserId` in an invalid way:

```
# '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 function that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce any overhead beyond that of a regular function call.

More precisely, the expression `some_value is Derived(some_value)` is always true at runtime.

This also means that it is not possible to create a subtype of `Derived` since it is an identity function at runtime, not an actual type:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
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)
```

and typechecking for `ProUserId` will work as expected.

See **PEP 484** for more details.

참고: Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing `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.

In contrast, `NewType` declares one type to be a *subtype* of another. Doing `Derived = NewType('Derived', Original)` will make the static type checker treat `Derived` as a *subclass* of `Original`, which means a value of type `Original` cannot be used in places where a value of type `Derived` is expected. This is useful when you want to prevent logic errors with minimal runtime cost.

버전 3.5.2에 추가.

27.1.3 Callable

Frameworks expecting callback functions of specific signatures might be type hinted using `Callable[[Arg1Type, Arg2Type], ReturnType]`.

For example:

```
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # Body
```

It is possible to declare the return type of a callable without specifying the call signature by substituting a literal ellipsis for the list of arguments in the type hint: `Callable[..., ReturnType]`.

27.1.4 Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, abstract base classes have been extended to support subscription to denote expected types for container elements.

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generics can be parameterized by using a new factory available in typing called *TypeVar*.

```
from typing import Sequence, TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

27.1.5 User-defined generic types

A user-defined class can be defined as a generic class.

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[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)
```

`Generic[T]` as a base class defines that the class `LoggedVar` takes a single type parameter `T` . This also makes `T` valid as a type within the class body.

The *Generic* base class defines `__class_getitem__()` so that `LoggedVar[t]` is valid as a type:

```
from typing 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, and type variables may be constrained:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

Each type variable argument to *Generic* must be distinct. This is thus invalid:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

You can use multiple inheritance with *Generic*:

```
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

When inheriting from generic classes, some type variables could be fixed:

```
from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

In this case `MyDict` has a single parameter, `T`.

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 typing import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

User defined generic type aliases are also supported. Examples:

```
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]

def inproduct(v: Vec[T]) -> T: # Same as Iterable[Tuple[T, T]]
    return sum(x*y for x, y in v)
```

버전 3.7에서 변경: *Generic* no longer has a custom metaclass.

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.

27.1.6 The `Any` type

A special kind of type is *Any*. A static type checker will treat every type as being compatible with *Any* and *Any* as being compatible with every type.

This means that it is possible to perform any operation or method call on a value of type on *Any* and assign it to any variable:

```
from typing import Any

a = None     # type: Any
a = []       # OK
a = 2        # OK

s = ''       # type: str
s = a        # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no typechecking 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!

Furthermore, all functions without a return type or parameter types will implicitly default to using *Any*:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

This behavior allows *Any* to be used as an *escape hatch* when you need to mix dynamically and statically typed code.

Contrast the behavior of *Any* with the behavior of *object*. Similar to *Any*, every type is a subtype of *object*. However, unlike *Any*, the reverse is not true: *object* is *not* a subtype of every other type.

That means when the type of a value is *object*, a type checker will reject almost all operations on it, and assigning it to a variable (or using it as a return value) of a more specialized type is a type error. For example:

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Use *object* to indicate that a value could be any type in a typesafe manner. Use *Any* to indicate that a value is dynamically typed.

27.1.7 Classes, functions, and decorators

The module defines the following classes, functions and decorators:

class typing.**TypeVar**

Type variable.

Usage:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be 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 definitions. See class Generic for more information on generic types. Generic functions work as follows:

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

The latter example's signature is essentially the overloading of `(str, str) -> str` and `(bytes, bytes) -> bytes`. Also note that if the arguments are instances of some subclass of *str*, the return type is still plain *str*.

At runtime, `isinstance(x, T)` will raise *TypeError*. In general, *isinstance()* and *issubclass()* should not be used with types.

Type variables may be marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. See **PEP 484** for more details. By default type variables are invariant. Alternatively, a type variable may specify an upper bound using `bound=<type>`. This means that an actual type substituted (explicitly or implicitly) for the type variable must be a subclass of the boundary type, see **PEP 484**.

class typing.**Generic**

Abstract base class for generic types.

A generic type is typically declared by inheriting from an instantiation of this class with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

This class can then be used as follows:

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class typing.**Type**(*Generic[CT_co]*)

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `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 BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

The fact that `Type[C]` is covariant implies that all subclasses of `C` should implement the same constructor signature and class method signatures as `C`. The type checker should flag violations of this, but should also allow constructor calls in subclasses that match the constructor calls in the indicated base class. How the type checker is required to handle this particular case may change in future revisions of **PEP 484**.

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[Union[BaseUser, ProUser]]): ...
```

`Type[Any]` is equivalent to `Type` which in turn is equivalent to `type`, which is the root of Python's metaclass hierarchy.

버전 3.5.2에 추가.

class typing.**Iterable**(*Generic[T_co]*)
A generic version of *collections.abc.Iterable*.

class typing.**Iterator**(*Iterable[T_co]*)
A generic version of *collections.abc.Iterator*.

class typing.**Reversible**(*Iterable[T_co]*)
A generic version of *collections.abc.Reversible*.

class typing.**SupportsInt**
An ABC with one abstract method `__int__`.

class typing.**SupportsFloat**
An ABC with one abstract method `__float__`.

class typing.**SupportsComplex**
An ABC with one abstract method `__complex__`.

class typing.**SupportsBytes**
An ABC with one abstract method `__bytes__`.

class typing.**SupportsAbs**
An ABC with one abstract method `__abs__` that is covariant in its return type.

class typing.**SupportsRound**
An ABC with one abstract method `__round__` that is covariant in its return type.

class typing.**Container**(*Generic[T_co]*)
A generic version of *collections.abc.Container*.

class typing.**Hashable**
An alias to *collections.abc.Hashable*

class typing.**Sized**
An alias to *collections.abc.Sized*

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)
A generic version of *collections.abc.Collection*

버전 3.6.0에 추가.

class typing.**AbstractSet**(*Sized, Collection[T_co]*)
A generic version of *collections.abc.Set*.

class typing.**MutableSet**(*AbstractSet[T]*)
A generic version of *collections.abc.MutableSet*.

class typing.**Mapping**(*Sized, Collection[KT], Generic[VT_co]*)
A generic version of *collections.abc.Mapping*. This type can be used as follows:

```
def get_position_in_index(word_list: Mapping[str, int], word: str) -> int:
    return word_list[word]
```

class typing.**MutableMapping**(*Mapping[KT, VT]*)
A generic version of *collections.abc.MutableMapping*.

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)
A generic version of *collections.abc.Sequence*.

class typing.**MutableSequence**(*Sequence[T]*)
A generic version of *collections.abc.MutableSequence*.

class typing.**ByteString**(*Sequence[int]*)
A generic version of *collections.abc.ByteString*.

This type represents the types *bytes*, *bytearray*, and *memoryview*.

As a shorthand for this type, *bytes* can be used to annotate arguments of any of the types mentioned above.

class typing.**Deque**(*deque, MutableSequence[T]*)
A generic version of *collections.deque*.

버전 3.5.4에 추가.

버전 3.6.1에 추가.

class typing.**List**(*list, MutableSequence[T]*)
Generic version of *list*. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as *Sequence* or *Iterable*.

This type may be used as follows:

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

class typing.**Set**(*set, MutableSet[T]*)
A generic version of *builtins.set*. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as *AbstractSet*.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)
A generic version of *builtins.frozenset*.

class typing.**MappingView**(*Sized, Iterable[T_co]*)
A generic version of *collections.abc.MappingView*.

class typing.**KeysView**(*MappingView[KT_co], AbstractSet[KT_co]*)
A generic version of *collections.abc.KeysView*.

class typing.**ItemsView**(*MappingView, Generic[KT_co, VT_co]*)
A generic version of *collections.abc.ItemsView*.

class typing.**ValuesView**(*MappingView[VT_co]*)
A generic version of *collections.abc.ValuesView*.

class typing.**Awaitable**(*Generic[T_co]*)
A generic version of *collections.abc.Awaitable*.

버전 3.5.2에 추가.

class typing.**Coroutine**(*Awaitable[V_co], Generic[T_co T_contra, V_co]*)
A generic version of *collections.abc.Coroutine*. The variance and order of type variables correspond to those of *Generator*, for example:

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

버전 3.5.3에 추가.

class typing.**AsyncIterable**(*Generic[T_co]*)
A generic version of *collections.abc.AsyncIterable*.

버전 3.5.2에 추가.

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)
A generic version of *collections.abc.AsyncIterator*.

버전 3.5.2에 추가.

class typing.**ContextManager**(*Generic[T_co]*)
A generic version of *contextlib.AbstractContextManager*.

버전 3.5.4에 추가.

버전 3.6.0에 추가.

class typing.**AsyncContextManager**(*Generic[T_co]*)
A generic version of *contextlib.AbstractAsyncContextManager*.

버전 3.5.4에 추가.

버전 3.6.2에 추가.

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)
A generic version of *dict*. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as *Mapping*.

This type can be used as follows:

```
def count_words(text: str) -> Dict[str, int]:
    ...
```

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)
A generic version of *collections.defaultdict*.

버전 3.5.2에 추가.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)
A generic version of *collections.OrderedDict*.

버전 3.7.2에 추가.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)
A generic version of *collections.Counter*.

버전 3.5.4에 추가.

버전 3.6.1에 추가.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)
A generic version of *collections.ChainMap*.

버전 3.5.4에 추가.

버전 3.6.1에 추가.

class typing.**Generator**(*Iterator[T_co], Generic[T_co, T_contra, V_co]*)

A generator can be annotated by 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 generics in the typing module, the `SendType` of *Generator* behaves contravariantly, not covariantly or invariantly.

If your generator will only yield values, set the `SendType` and `ReturnType` to `None`:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Alternatively, annotate your generator 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
```

class typing.**AsyncGenerator**(*AsyncIterator[T_co], Generic[T_co, T_contra]*)

An async generator can be annotated by the generic type `AsyncGenerator[YieldType, SendType]`. For example:

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

Unlike normal generators, async generators cannot return a value, so there is no `ReturnType` type parameter. As with *Generator*, the `SendType` behaves contravariantly.

If your generator will only yield values, set the `SendType` to `None`:

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

Alternatively, annotate your generator as having a return type of either `AsyncIterable[YieldType]` or `AsyncIterator[YieldType]`:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

버전 3.6.1에 추가.

class typing.**Text**

`Text` is an alias for `str`. It is provided to supply a forward compatible path for Python 2 code: in Python 2, `Text` is an alias for `unicode`.

Use `Text` to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

버전 3.5.2에 추가.

class typing.**IO**
class typing.**TextIO**
class typing.**BinaryIO**

Generic type `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by *open()*.

class typing.**Pattern**
class typing.**Match**

These type aliases correspond to the return types from *re.compile()* and *re.match()*. These types (and the corresponding functions) are generic in `AnyStr` and can be made specific by writing `Pattern[str]`, `Pattern[bytes]`, `Match[str]`, or `Match[bytes]`.

class typing.**NamedTuple**

Typed version of *collections.namedtuple()*.

Usage:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Fields with a default value must come after any fields without a default.

The resulting class has two extra attributes: `_field_types`, giving a dict mapping field names to types, and `_field_defaults`, a dict mapping field names to default values. (The field names are in the `_fields` attribute, which is part of the namedtuple API.)

`NamedTuple` subclasses can also have docstrings and methods:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

Backward-compatible usage:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

버전 3.6에서 변경: Added support for **PEP 526** variable annotation syntax.

버전 3.6.1에서 변경: Added support for default values, methods, and docstrings.

class typing.**ForwardRef**

A class used for internal typing representation of string forward references. For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. This class should not be instantiated by a user, but may be used by introspection tools.

typing.**NewType**(*typ*)

A helper function to indicate a distinct types to a typechecker, see *NewType*. At runtime it returns a function that returns its argument. Usage:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

버전 3.5.2에 추가.

typing.**cast**(*typ*, *val*)

Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

typing.**get_type_hints**(*obj*[, *globals*[, *locals*]])

Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`. In addition, forward references encoded as string literals are handled by evaluating them in `globals` and `locals` namespaces. If necessary, `Optional[t]` is added for function and method annotations if a default value equal to `None` is set. For a class `C`, return a dictionary constructed by merging all the `__annotations__` along `C.__mro__` in reverse order.

@typing.**overload**

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). The `@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition, while the latter is used at runtime but should be ignored by a type checker. At runtime, calling a `@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:
    ...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def process(response):
    <actual implementation>
```

See **PEP 484** for details and comparison with other typing semantics.

`@typing.`**`no_type_check`**
Decorator to indicate that annotations are not type hints.

This works as class or function *decorator*. With a class, it applies recursively to all methods defined in that class (but not to methods defined in its superclasses or subclasses).

This mutates the function(s) in place.

`@typing.`**`no_type_check_decorator`**
Decorator to give another decorator the *no_type_check()* effect.

This wraps the decorator with something that wraps the decorated function in *no_type_check()*.

`@typing.`**`type_check_only`**
Decorator to mark a class or function to be unavailable at runtime.

This decorator is itself not available at runtime. It is mainly intended to mark classes that are defined in type stub files if an implementation returns an instance of a private class:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Note that returning instances of private classes is not recommended. It is usually preferable to make such classes public.

`typing.`**`Any`**
Special type indicating an unconstrained type.

- Every type is compatible with *Any*.
- *Any* is compatible with every type.

`typing.`**`NoReturn`**
Special type indicating that a function never returns. For example:

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

버전 3.5.4에 추가.

버전 3.6.2에 추가.

`typing.`**`Union`**
Union type; `Union[X, Y]` means either X or Y.

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- You cannot subclass or instantiate a union.
- You cannot write `Union[X][Y]`.
- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

버전 3.7에서 변경: Don't remove explicit subclasses from unions at runtime.

typing.**Optional**

Optional type.

`Optional[X]` is equivalent to `Union[X, None]`.

Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example:

```
def foo(arg: int = 0) -> None:
    ...
```

On the other hand, if an explicit value of `None` is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

typing.**Tuple**

Tuple type; `Tuple[X, Y]` is the type of a tuple of two items with the first item of type X and the second of type Y. The type of the empty tuple can be written as `Tuple[()]`.

Example: `Tuple[T1, T2]` is a tuple of two elements corresponding to type variables T1 and T2. `Tuple[int, float, str]` is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use literal ellipsis, e.g. `Tuple[int, ...]`. A plain *Tuple* is equivalent to `Tuple[Any, ...]`, and in turn to *tuple*.

typing.**Callable**

Callable type; `Callable[[int], str]` is a function of (int) -> str.

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 or an ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments; such function types are rarely used as callback types. `Callable[..., ReturnType]` (literal ellipsis) can be used to type hint a callable taking any number of arguments and returning `ReturnType`. A plain *Callable* is equivalent to `Callable[..., Any]`, and in turn to *collections.abc.Callable*.

typing.**ClassVar**
Special type construct to mark class variables.

As introduced in **PEP 526**, a variable annotation wrapped in ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. Usage:

```
class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

ClassVar accepts only types and cannot be further subscribed.

ClassVar is not a class itself, and should not be used with *isinstance()* or *issubclass()*. *ClassVar* does not change Python runtime behavior, but it can be used by third-party type checkers. For example, a type checker might flag the following code as an error:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

버전 3.5.3에 추가.

typing.**AnyStr**
AnyStr is a type variable defined as AnyStr = TypeVar('AnyStr', str, bytes).

It is meant to be used for functions that may accept any kind of string without allowing different kinds of strings to mix. For example:

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

typing.**TYPE_CHECKING**
A special constant that is assumed to be True by 3rd party static type checkers. It is False at runtime. Usage:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

Note that the first type annotation must be enclosed in quotes, making it a "forward reference", to hide the expensive_mod reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

버전 3.5.2에 추가.

27.2 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, *pydoc*은 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 *pydoc*을 사용하여 설명서를 콘솔에 텍스트로 생성합니다. 같은 텍스트 설명서는 운영 체제의 명령 프롬프트에서 **pydoc**을 스크립트로 실행하여 파이썬 인터프리터 외부에서도 볼 수 있습니다. 예를 들어,

```
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, *pydoc*은 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. `PAGER` 환경 변수가 설정되면, **pydoc**은 그 값을 페이지 매김 프로그램으로 사용합니다.

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. **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.

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.

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.

27.3 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:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 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

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    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:  # catch a value like 1e300
        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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
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.py`에서 찾을 수 있습니다.

27.3.1 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.

`verbose=True`를 *testmod()*에 전달하여 상세 모드를 강제하거나, `verbose=False`를 전달하여 상세 모드를 금지할 수 있습니다. 두 경우 모두, `sys.argv`는 *testmod()*에 의해 검사되지 않습니다 (따라서 `-v`를 전달하거나 그렇지 않아도 효과가 없습니다).

또한 *testmod()*를 실행하는 명령 줄 단축법이 있습니다. 파이썬 인터프리터에게 표준 라이브러리에서 직접 doctest 모듈을 실행하도록 지시하고 명령 줄에 모듈 이름(들)을 전달할 수 있습니다:

```
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.

*testmod()*에 대한 자세한 내용은, 기본 *API* 절을 참조하십시오.

27.3.2 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.txt에 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

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
```

*testmod()*와 마찬가지로, *testfile()*은 예제가 실패하지 않는 한 아무것도 표시하지 않습니다. 예제가 실패하면, 실패한 예제와 실패 원인이 *testmod()*와 같은 형식을 사용하여 stdout에 인쇄됩니다.

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.

또한 *testfile()*를 실행하는 명령 줄 단축법이 있습니다. 파이썬 인터프리터에게 표준 라이브러리에서 직접 doctest 모듈을 실행하도록 지시하고 명령 줄에 파일 이름(들)을 전달할 수 있습니다:

```
python -m doctest -v example.txt
```

파일 이름은 .py로 끝나지 않으므로, *doctest*는 *testmod()*가 아니라 *testfile()*로 실행되어야 한다고 추론합니다.

*testfile()*에 대한 자세한 내용은, 기본 *API* 절을 참조하십시오.

27.3.3 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, `M.__test__`가 존재하고 "참이면", 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 이름으로 나타납니다

```
<name of M>.__test__.K
```

발견된 모든 클래스는 포함된 메서드와 중첩된 클래스의 독스트링을 테스트하기 위해 유사하게 재귀적으로 검색됩니다.

CPython implementation detail: Prior to version 3.4, extension modules written in C were not fully searched by doctest.

독스트링 예제는 어떻게 인식됩니까?

대부분 대화형 콘솔 세션의 복사하여 붙여넣기가 잘 작동하지만, doctest는 특정 파이썬 셸의 정확한 에뮬레이션을 시도하지 않습니다.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
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 발생

하지 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_dict를 *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
```

[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.

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:

- 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.
- 대화형 셸은 일부 *SyntaxError*에서 트레이스백 헤더 줄을 생략합니다. 그러나 doctest는 트레이스백 헤더 줄을 사용하여 예외를 비 예외와 구별합니다. 따라서 트레이스백 헤더를 생략하는 *SyntaxError*를 테스트해야 하는 드문 경우에는, 트레이스백 헤더 줄을 수동으로 테스트 예제에 추가해야 합니다.
- 일부 *SyntaxError*의 경우, 파이썬은 ^ 마커를 사용하여 구문 에러의 문자 위치를 표시합니다:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
      ^
SyntaxError: invalid syntax
```

에러의 위치를 나타내는 줄은 예외 형과 세부 사항 앞에 오므로, doctest가 점검하지 않습니다. 예를 들어, ^ 마커를 잘못된 위치에 넣어도, 다음 테스트가 통과합니다:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
    ^
SyntaxError: invalid syntax
```

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.

버전 3.4에 추가: `-o` 명령 줄 옵션.

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**
: 지정하면, 예외를 예상하는 예제가, 예외 세부 사항이 일치하지 않아도 예상 형의 예외가 발생하면 통과합니다. 예를 들어, `ValueError: 42`를 예상하는 예제는 발생한 실제 예외가 `ValueError: 3*14`이면 통과하지만, 예를 들어 *TypeError*가 발생하면 실패합니다.

  또한, 파이썬 3 doctest 보고서에 사용된 모듈 이름도 무시합니다. 따라서 이 두 변형은 이 플래그가 지정되면 테스트가 파이썬 2.7이나 파이썬 3.2(또는 이후 버전)에서 실행되는지와 관계없이 작동합니다:

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

*ELLIPSIS*를 사용하여 예외 메시지의 세부 사항을 무시할 수도 있지만, 그러한 테스트는 모듈 세부 사항이 예외 이름의 일부로 인쇄되는지에 따라 여전히 실패할 수 있음에 유의하십시오. *IGNORE_EXCEPTION_DETAIL*과 파이썬 2.3의 세부 사항을 사용하는 것은 또한 예외 세부 사항에 신경 쓰지 않고 여전히 파이썬 2.3이나 그 이전 버전(이 배포는 *doctest 지시자*를 지원하지 않고 무의미한 주석으로 무시합니다)에서 통과하는 doctest를 작성하는 유일하게 명확한 방법입니다. 예를 들면:

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

는 플래그가 지정될 때 파이썬 2.3 이후 버전에서 통과합니다. 파이썬 2.4에서 세부 사항이 "doesn't" 대신 "does not"으로 변경되었음에도 통과합니다.

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 명령 줄은 옵션 `-f`를 `-o FAIL_FAST`의 축약으로 받아들입니다.

버전 3.4에 추가.

`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)))
[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)))
[0, 1, ..., 18, 19]
```

하나의 물리적 줄에 여러 개의 지시자를 쉼표로 구분하여 사용할 수 있습니다:

```
>>> print(list(range(20)))
[0,    1, ...,   18,    19]
```

하나의 예제에 여러 개의 지시자 주석이 사용되면, 모두 결합합니다:

```
>>> print(list(range(20)))
...
[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)))
...
[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()
{"Hermione", "Harry"}
```

취약합니다! 한 가지 해결 방법은 다음과 같습니다

```
>>> foo() == {"Hermione", "Harry"}
True
```

대신에. 또 다른 방법은

```
>>> d = sorted(foo())
>>> d
['Harry', 'Hermione']
```

참고: 파이썬 3.6 이전에는, 딕셔너리를 인쇄할 때, 파이썬은 키-값 쌍이 특정 순서로 인쇄되는 것을 보증하지 않았습니다.

다른 것들도 있지만, 아마 아이디어를 얻었을 겁니다.

또 다른 나쁜 생각은 객체 주소를 포함하는 것들을 출력하는 것입니다

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()   # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

ELLIPSIS 지시자는 마지막 예제를 다루는 좋은 접근법을 제공합니다

```
>>> C()
<__main__.C instance at 0x...>
```

부동 소수점 숫자도 플랫폼에 따라 약간의 출력 변동이 있습니다. 파이썬이 float 포매팅을 플랫폼 C 라이브러리에 위임하고 있고, 이때 C 라이브러리의 품질이 크게 다르기 때문입니다.

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

`I/2.**J` 형식의 숫자는 모든 플랫폼에서 안전하며, 저는 종종 이런 형식의 숫자를 만들도록 doctest 예제를 꾸밉니다:

```
>>> 3./4  # utterly safe
0.75
```

간단한 분수는 또한 사람들이 이해하기가 더 쉬우므로, 더 좋은 설명서가 되도록 합니다.

27.3.4 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.

선택적 인자 *verbose*가 참이면 많은 것들을 인쇄하고, 거짓이면 실패만 인쇄합니다; 기본적으로, 또는 None이면, '-v'가 sys.argv에 있을 때만 참입니다.

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).

선택적 인자 *optionflags*(기본값은 0)는 옵션 플래그의 비트별 OR를 취합니다. 옵션 플래그 절을 참조하십시오.

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 None이면 모듈 *__main__*)에서 도달할 수 있는 함수와 클래스의 독스트링에 있는 예제를 테스트합니다. m.__doc__으로 시작합니다.

딕셔너리 m.__test__이 존재하고 None이 아니면, 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 None이면, 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 None이면, *globs*에서 발견되는 퓨처 기능 집합에 해당하는 플래그가 추론됩니다.

선택적 인자 *optionflags*는 위의 함수 *testfile()*에서 처럼 작동합니다.

27.3.5 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.

반환된 *unittest.TestSuite*는 unittest 프레임워크에 의해 실행되고, 각 파일에 있는 대화식 예제를 실행합니다. 어떤 파일의 예제가 실패하면, 합성된 단위 테스트가 실패하고, 테스트를 포함하는 파일의 이름과 (때로는 근사치인) 줄 번호를 보여주는 `failureException` 예외가 발생합니다.

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.

선택적 인자 *setUp*은 테스트 스위트에 대한 사전 설정(set-up) 함수를 지정합니다. 이것은 각 파일에서 테스트를 실행하기 전에 호출됩니다. *setUp* 함수로 *DocTest* 객체가 전달됩니다. setUp 함수는 전달된 테스트의 *globs* 어트리뷰트를 통해 테스트 전역에 액세스할 수 있습니다.

선택적 인자 *tearDown*은 테스트 스위트에 사후 정리(tear-down) 함수를 지정합니다. 이것은 각 파일에서 테스트를 실행한 후에 호출됩니다. *tearDown* 함수로 *DocTest* 객체가 전달됩니다. tearDown 함수는 전달된 테스트의 *globs* 어트리뷰트를 통해 테스트 전역에 액세스할 수 있습니다.

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 OR해서 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*, *checker=None*)

모듈에 대한 doctest 테스트를 *unittest.TestSuite*로 변환합니다.

반환된 *unittest.TestSuite*는 unittest 프레임워크에 의해 실행되고, 모듈에 있는 각 doctest를 실행합니다. 어떤 doctest가 실패하면, 합성된 단위 테스트가 실패하고, 테스트를 포함하는 파일의 이름과 (때로는 근사치인) 줄 번호를 보여주는 `failureException` 예외가 발생합니다.

선택적 인자 *module*은 테스트할 모듈을 제공합니다. 모듈 객체나 (점으로 구분될 수 있는) 모듈 이름일 수 있습니다. 지정하지 않으면, 이 함수를 호출하는 모듈이 사용됩니다.

선택적 인자 *globs*는 테스트의 초기 전역 변수를 포함하는 딕셔너리입니다. 이 딕셔너리의 새 사본이 테스트마다 만들어집니다. 기본적으로, *globs*는 새로운 빈 딕셔너리입니다.

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 드롭 인 대체)입니다.

선택적 인자 *setUp*, *tearDown* 및 *optionflags*는 위의 함수 *DocFileSuite()*와 같습니다.

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.

따라서 *unittest.TestSuite*를 만드는 두 가지 방법 모두 `DocTestCase`의 인스턴스를 실행합니다. 이것은 미묘한 이유로 중요합니다: 여러분이 *doctest* 함수를 직접 실행할 때, 옵션 플래그를 *doctest* 함수에 전달하여 사용 중인 *doctest* 옵션을 직접 제어할 수 있습니다. 그러나, *unittest* 프레임워크를 작성한다면, *unittest*가 테스트가 언제 어떻게 실행되는지 궁극적으로 제어합니다. 프레임워크 저자는 일반적으로 *doctest* (아마도, 예를 들어, 명령 줄 옵션으로 지정하는) 보고(reporting) 옵션을 제어하려고 하지만, *unittest*를 통해 *doctest* 테스트 실행기로 옵션을 전달할 방법이 없습니다.

이러한 이유로, *doctest*는 *unittest* 지원에 특화된 *doctest* 보고(reporting) 플래그 개념을 다음 함수를 통해 지원합니다:

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.

이것은 모듈 전역 설정이며, 모듈 *unittest*에 의해 실행되는 모든 미래의 doctest에 영향을 줍니다. `DocTestCase`의 `runTest()` 메서드는 `DocTestCase` 인스턴스가 생성될 때 테스트 케이스에 대해 지정된 옵션 플래그를 봅니다. 보고 플래그가 지정되지 않았으면 (이것이 일반적이고 예상되는 경우입니다), *doctest*의 *unittest*보고 플래그는 옵션 플래그에 비트별 OR되고, 이렇게 손질된 옵션 플래그가 doctest를 실행하기 위해 만들어진 *DocTestRunner* 인스턴스로 전달됩니다. `DocTestCase` 인스턴스가 생성될 때 보고 플래그가 지정되었으면, *doctest*의 *unittest* 보고 플래그는 무시됩니다.

함수가 호출되기 전에 유효했던 *unittest* 보고 플래그의 값이 함수에 의해 반환됩니다.

27.3.6 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.

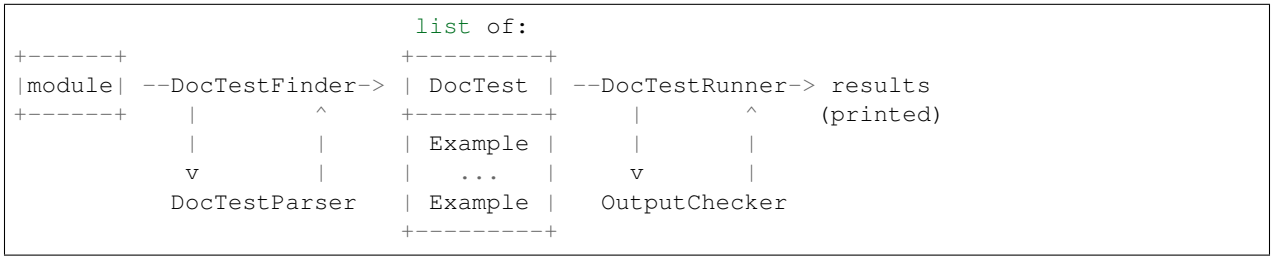
고급 API는 doctest 케이스에서 추출한 대화식 예제를 저장하는 데 사용되는 두 개의 컨테이너 클래스를 중심으로 돌아갑니다:

- *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:



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

이 *DocTest*가 추출된 파일의 이름; 또는 파일 이름을 알 수 없거나 파일에서 *DocTest*가 추출되지 않았으면 None.

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 반환 값과 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 optionflags에 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 `None`이면, 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 `False`면, 모듈을 찾으려고 시도하지 않습니다. 이것은 눈에 띄지 않는데, 대부분 doctest 자체를 테스트할 때 사용합니다: *module*이 `False`이거나, `None`이지만 자동으로 찾을 수 없으면, 모든 객체는 (존재하지 않는) 모듈에 속한 것으로 간주하므로, 포함된 모든 객체에서 (재귀적으로) doctest를 검색합니다.

각 *DocTest*에 대한 전역은 *globs*와 *extraglobs*(*extraglobs*의 바인딩이 *globs*의 바인딩에 우선합니다)를 결합하여 구성됩니다. 각 *DocTest*마다 전역 딕셔너리의 새 얕은 복사본이 만들어집니다. *globs*를 지정하지 않으면, 기본값은 모듈이 지정되었다면 모듈의 *__dict__*, 또는 그렇지 않으면 `{}`입니다. *extraglobs*가 지정되지 않으면, 기본값은 `{}`입니다.

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* 및 *lineno*는 새 *DocTest* 객체의 어트리뷰트입니다. 자세한 내용은 *DocTest* 설명서를 참조하십시오.

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>'*)
주어진 문자열을 예제와 중간에 있는 텍스트로 나누고, 이를 *Example*와 문자열이 번갈아 나오는 리스트로 반환합니다. *Example*의 줄 번호는 0부터 시작합니다. 선택적 인자 *name*은, 이 문자열을 식별하는 이름이며, 에러 메시지에만 사용됩니다.

DocTestRunner 객체

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

*DocTest*에 있는 대화형 예제를 실행하고 검증하는 데 사용되는 처리 클래스.

예상 출력과 실제 출력 간의 비교는 *OutputChecker*에 의해 수행됩니다. 이 비교는 여러 옵션 플래그로 사용자 정의할 수 있습니다; 자세한 내용은 옵션 플래그 절을 참조하십시오. 옵션 플래그로 충분하지 않으면, *OutputChecker*의 서브 클래스를 생성자에 전달하여 비교를 사용자 정의할 수도 있습니다.

테스트 실행기의 디스플레이 출력은 두 가지 방법으로 제어할 수 있습니다. 첫째, 출력 함수를 TestRunner.run() 로 전달할 수 있습니다; 이 함수는 표시되어야 하는 문자열로 호출됩니다. 기본값은 sys.stdout.write입니다. 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 True이면, 실행될 때 각 예제에 대한 정보가 인쇄됩니다. *verbose*가 False이면, 실패만 인쇄됩니다. *verbose*가 지정되지 않거나 None이면, 명령 줄 스위치 -v가 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.

*DocTestParser*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*은 처리될 예제입니다. *test*는 예제를 포함하는 테스트입니다. *out*은 *DocTestRunner.run()*에 전달된 출력 함수입니다.

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.

예제는 이름 공간 test.globs에서 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 실행한 모든 테스트 케이스의 요약을 인쇄하고, 네임드 튜플 `TestResults(failed, attempted)`를 반환합니다.

선택적 *verbose* 인자는 요약이 얼마나 상세할지를 제어합니다. 상세도가 지정되지 않으면, *DocTestRunner*의 상세도가 사용됩니다.

OutputChecker 객체

class doctest.**OutputChecker**
A class used to check the whether the actual output from a doctest example matches the expected output. *OutputChecker* defines two methods: *check_output()*, which compares a given pair of outputs, and returns `True` if they match; and *output_difference()*, which returns a string describing the differences between two outputs.

*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.

27.3.7 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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> f(3)
9
"""
```

그러면 대화형 파이썬 세션은 이런 식이 됩니다:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <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.

27.3.8 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_topic`이라는 함수를 정의하십시오. 이 함수들은 모듈과 같은 파일에 포함되거나 별도의 테스트 파일로 분리될 수 있습니다.
- 회귀 테스트 주제에서 테스트 케이스가 포함된 독스트링에 대한 `__test__` 딕셔너리 매핑을 정의하십시오.

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("{} failures out of {} tests".format(fail, total))
```

27.4 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 픽스처 테스트 픽스처 *(test fixture)*는 1개 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 Third-party unittest framework with a lighter-weight syntax for writing tests. For example, `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 또는 Hudson과 같은 지속적인 통합 시스템을 이용하여 테스트가 이루어지길 추천합니다.

27.4.1 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'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

테스트 케이스는 *unittest.TestCase*를 서브 클래스 해서 생성하였습니다. 각각 3개의 테스트는 `test` 글자로 시작하는 이름을 가진 메서드로 정의했습니다. 이 명명 규칙은 테스트 실행자가 어떤 메서드가 테스트인지 알게 해줍니다.

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:

```
...
----------------------------------------------------------------------
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) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... 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.

27.4.2 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.

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-C를 누르면 현재 테스트가 끝날 때까지 기다린 다음 지금까지의 모든 결과를 보고합니다. Control-C를 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 패턴과 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 foo는 foo_tests.SomeTest.test_something, bar_tests.SomeTest.test_foo에 일치하지만, bar_tests.FooTest.test_something에는 일치하지 않습니다.

--locals
트레이스백(traceback)에서 지역 변수를 표시합니다.

버전 3.2에 추가: 명령행 옵션인 -b, -c, -f가 추가되었습니다.

버전 3.5에 추가: 명령행 옵션 --locals.

버전 3.7에 추가: 명령행 옵션 -k.

명령행은 프로젝트의 모든 테스트 또는 일부분의 테스트 탐색을 위해서도 사용할 수 있습니다.

27.4.3 테스트 탐색(Discovery)

버전 3.2에 추가.

unittest는 간단한 테스트 탐색을 지원합니다. 테스트 탐색에 호환되기 위해서는 모든 테스트 파일은 반드시 프로젝트의 가장 상위 디렉터리로부터 모듈 또는 패키지(이름 공간 패키지 포함)로 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.test`를 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.py`는 `foo.bar.baz`로 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: 테스트 탐색은 이름 공간 패키지를 지원합니다.

27.4.4 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 `test`로 이름이 시작하는 함수)를 구현하는 것입니다:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    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 픽스처*(test fixture)*라고 부릅니다. 개별 테스트 메서드를 실행하기 위해 고유한 테스트 픽스처에 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 file:*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.

27.4.5 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.

27.4.6 테스트 건너뛰기와 예상된 실패

버전 3.1에 추가.

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):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

아래는 위의 예를 상세 모드로 실행했을 때의 출력입니다:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase) ... skipped 'external resource not available'
test_windows_support (__main__.MyTestCase) ... 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 실패라는 표시를 합니다. 테스트가 실패하면 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.

27.4.7 부분 테스트(subtest)를 사용하여 테스트 반복 구별 짓기

버전 3.4에 추가.

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):
        """
        Test that numbers between 0 and 5 are all even.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

다음의 출력을 만듭니다:

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, 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)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

27.4.8 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 픽스처를 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 픽스처를 보십시오.

버전 3.2에 추가.

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.

버전 3.2에 추가.

run(*result=None*)
테스트를 실행하고, *result* 인자로 전달된 *TestResult*에 결과를 수집합니다. 만약 *result* 인자가 전달 안 되거나 None이라면 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 run은 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.

버전 3.1에 추가.

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.

버전 3.4에 추가.

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

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 객체로 평가되는지(아닌지) 테스트합니다.

버전 3.1에 추가.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)
*expr*이`None` 인지(아닌지) 테스트합니다.

버전 3.1에 추가.

assertIn(*member*, *container*, *msg=None*)
assertNotIn(*member*, *container*, *msg=None*)
Test that *member* is (or is not) in *container*.

버전 3.1에 추가.

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.

버전 3.2에 추가.

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

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')
```

버전 3.1에 추가: `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.

버전 3.2에 추가.

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')
```

버전 3.2에 추가.

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 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')
   logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

버전 3.4에 추가.

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

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"
```

버전 3.1에 추가.

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.

버전 3.1에 추가: `assertRegexpMatches` 라는 이름으로 추가되었습니다.

버전 3.2에서 변경: `assertRegexpMatches()` 메서드가 *assertRegex()*로 이름 변경되었습니다.

버전 3.2에 추가: *assertNotRegex()*.

버전 3.5에 추가: `assertNotRegexpMatches`이름은 *assertNotRegex()*의 폐지된 에일리어스입니다.

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.

버전 3.2에 추가.

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.

버전 3.1에 추가.

*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.

버전 3.1에 추가.

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 인스

턴스이어야 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.

버전 3.1에 추가.

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.

버전 3.1에 추가.

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.

버전 3.1에 추가.

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.

버전 3.1에 추가.

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.

버전 3.1에 추가.

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.

버전 3.2에 추가.

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 1입니다.

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 서브클래스는 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 None을 반환합니다. 이 메서드의 기본 구현은 가능하다면 테스트 메서드의 독스트링의 첫 번째 줄을 반환하고 그렇지 않으면 None을 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.

버전 3.1에 추가.

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.

버전 3.1에 추가.

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.

폐지된 에일리어스

역사적인 이유로 인해 *TestCase* 메서드의 일부는 지금은 폐지된 에일리어스를 1개 또는 그 이상 가졌습니다. 다음 표는 폐지된 에일리어스과 그에 맞는 올바른 이름을 나열합니다:

메서드 이름	폐지된 에일리어스	폐지된 에일리어스
assertEqual()	failUnlessEqual	assertEquals
assertNotEqual()	failIfEqual	assertNotEquals
assertTrue()	failUnless	assert_
assertFalse()	failIf	
assertRaises()	failUnlessRaises	
assertAlmostEqual()	failUnlessAlmostEqual	assertAlmostEquals
assertNotAlmostEqual()	failIfAlmostEqual	assertNotAlmostEquals
assertRegex()		assertRegexpMatches
assertNotRegex()		assertNotRegexpMatches
assertRaisesRegex()		assertRaisesRegexp

버전 3.1부터 폐지: 두 번째 열에 나열된 fail* 에일리어스는 폐지되었습니다.

버전 3.2부터 폐지: 세 번째 열에 나열된 assert* 에일리어스는 폐지되었습니다.

버전 3.2부터 폐지: `assertRegexpMatches`와 `assertRaisesRegexp`는 *assertRegex()*와 *assertRaisesRegex()*로 이름이 변경되었습니다.

버전 3.5부터 폐지: `assertNotRegexpMatches` 이름은 폐지되고 *assertNotRegex()*으로 대체합니다.

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 TestRunner에 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.

버전 3.5에 추가.

TestLoader 객체는 다음 메서드를 가집니다:

loadTestsFromTestCase(*testCaseClass*)
*TestCase*에서 파생된 testCaseClass에 포함된 모든 테스트 케이스의 묶음을 반환합니다.

테스트 케이스 인스턴스는 *getTestCaseNames()*에 의해 이름 지어진 각 메서드를 위해 생성됩니다. 기본값은 test로 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 프로토콜*입니다. *pattern* 인자는 `load_tests`에 세 번째 인자로 전달됩니다.

버전 3.2에서 변경: `load_tests` 지원이 추가됨.

버전 3.5에서 변경: 문서로 만들어 져 있지 않고 공식적이지 않은 *use_load_tests* 기본 인자를 폐지하고 무시합니다, 하지만 하위 호환성을 위해 여전히 그것을 수용합니다. 이 메서드는 이제 `load_tests`에 세 번째 인자로 전달되는 *pattern*을 오직 키워드 인자로써 수용합니다.

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. 반환 값은 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 최상위 디렉터리를 따로 지정해야 합니다.

만약 모듈 임포트가 실패한다면, 예를 들어 문법 에러로 인해, 이것은 1 개의 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.discover를 호출할지라도, 테스트 탐색은 실행 중에 패키지에 대한 테스트 검사를 한 번만 실행하도록 보장합니다.

만약 load_tests가 존재한다면 탐색은 패키지 안을 재귀 탐색하지 않습니다. load_tests가 패키지 안의 모든 테스트를 로드할 책임이 있습니다.

패턴은 의도적으로 로더 어트리뷰트로 저장되지 않아 패키지가 자신에 대한 탐색을 계속할 수 있습니다. *top_level_dir*는 저장되어 load_tests가 loader.discover()에게 이 인자를 건네줄 필요가 없습니다.

*start_dir*는 디렉터리뿐만 아니라 점으로 구분된 모듈 이름이 될 수 있습니다.

버전 3.2에 추가.

버전 3.4에서 변경: 임포트 시에 *SkipTest*가 발생한 모듈은 에러가 아닌, 건너뛰기로 기록됩니다. 탐색은 이름 공간 패키지를 지원합니다. 임포트되기 전에 경로들이 정렬되어 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_tests를 검사합니다, 왜냐하면 패키지 이름이 기본 패턴과 일치하는 것이 불가능하기 때문입니다.

*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(-v 옵션을 보십시오).

만약 이 어트리뷰트가 None(기본값)이 아니라면, 테스트 묶음에 포함될 모든 테스트 메서드는 이 목록의 패턴 중 1 개와 반드시 일치해야 합니다. 이 패턴 일치는 항상 *fnmatch.fnmatchcase()*를 사용하여 수행된다는 것을 기억하십시오, 그래서 -v 옵션에 패턴을 건네주는 것과 달리, 간단한 부분 문자열 패턴은 * 와일드카드를 사용하도록 변경되어야 할 것입니다.

이것은 모든 loadTestsFrom*() 메서드에 영향을 미칩니다.

버전 3.7에 추가.

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 `TestCase.assert*()` 메서드를 사용하여 명시적으로 실패가 발생한 테스트에 해당합니다.

skipped
TestCase 인스턴스와 테스트 건너뛰기한 이유 문자열로 구성된 2-튜플을 포함하는 목록입니다.

버전 3.1에 추가.

expectedFailures
TestCase 인스턴스와 포맷된(formatted) 트레이스백 문자열로 구성된 2-튜플을 포함하는 목록입니다. 각 튜플은 테스트 케이스의 예상된 실패에 해당합니다.

unexpectedSuccesses
예상된 실패로 표시되었지만 성공한 *TestCase* 인스턴스를 포함하는 목록입니다.

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.

버전 3.2에 추가.

failfast
참으로 설정하면 첫 번째 실패 또는 에러에서 *stop()*이 호출될 것입니다.

버전 3.2에 추가.

tb_locals
참으로 설정하면 지역 변수가 트레이스백에 보일 것입니다.

버전 3.5에 추가.

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 요구사항을 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.

버전 3.1에 추가.

stopTestRun()
모든 테스트가 실행되고 나서 1번 호출됩니다.

버전 3.1에 추가.

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 실패했지만 *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.

버전 3.4에 추가.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*)
*TextTestRunner*에서 사용하는 *TestResult*의 구체적인 구현입니다.

버전 3.2에 추가: 이 클래스는 이전에 `_TextTestResult` 이름이었습니다. 이 이름은 여전히 에일리어스로 존재하지만 폐지된 상태입니다.

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*)
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. 폐지된 *unittest* 메서드에 의해 발생한 폐지 경고도 특수한 경우이고, 경고 필터가 `'default'` 또는 `'always'` 일 때, 너무 많은 경고 메시지를 피하고자 그것들이 모듈당 1번만 보일 것입니다. 파이썬의 `-Wd`이나 `-Wa` 옵션(경고 제어를 보십시오)을 사용하고 *warnings*를 `None`으로 설정하여 이 동작을 오버라이드 할 수 있습니다.

버전 3.2에서 변경: `warnings` 인자 추가.

버전 3.2에서 변경: 임포트 시간이 아닌 인스턴스화 시간에 기본 스트림이 *sys.stderr*으로 설정됩니다.

버전 3.5에서 변경: tb_locals 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 None이면서 *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 None이면, *sys.argv* 값이 사용됩니다.

testRunner 인자는 테스트 실행자 클래스나 이미 생성된 테스트 실행자 인스턴스일 수 있습니다. 기본적으로 main은 실행한 테스트가 성공인지 실패인지를 나타내는 종료 코드와 함께 *sys.exit()*을 호출합니다.

testLoader 인자는 *TestLoader* 인스턴스이어야 하고 기본값은 *defaultTestLoader* 입니다.

main은 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 None으로 남아 있고, 그렇지 않은 경우에는 'default'로 설정됩니다.

사실 main 호출은 TestProgram 클래스의 인스턴스를 반환합니다. 이것은 실행된 테스트의 결과를 result 어트리뷰트에 저장합니다.

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

버전 3.2에 추가.

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_tests를 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_tests`를 위해 패키지 `__init__.py`를 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_tests`에게 맡겨질 것입니다:

```
load_tests(loader, standard_tests, pattern)
```

이것은 패키지의 모든 테스트에 해당하는 *TestSuite*를 반환해야 합니다. (`standard_tests`는 오직 `__init__.py`로부터 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):
    # top level directory cached on loader instance
    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.

27.4.9 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 `tearDownModule`이 실행되고, 이어 새로운 모듈의 `setUpModule`이 호출됩니다.

모든 테스트가 실행된 뒤에 마지막으로 `tearDownClass`와 `tearDownModule`이 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 픽스처 함수는 1번의 테스트 실행에서 여러 번 호출될 수 있습니다.

공유 픽스처는 비표준 정렬 순서를 사용하는 테스트 묶음과 같이 작동하는 것을 의도하지 않습니다. 공유 픽스처를 지원하길 원치 않는 프레임워크를 위해서 BaseTestSuite가 여전히 존재합니다.

공유 픽스처 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 setUpClass와 tearDownClass를 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 tearDownClass가 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.

27.4.10 시그널 처리하기

버전 3.2에 추가.

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):
    ...
```

27.5 unittest.mock — mock object library

버전 3.3에 추가.

Source code: Lib/unittest/mock.py

---

unittest.mock is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

unittest.mock provides a core *Mock* class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.

Additionally, mock provides a *patch()* decorator that handles patching module and class level attributes within the scope of a test, along with *sentinel* for creating unique objects. See the *quick guide* for some examples of how to use *Mock*, *MagicMock* and *patch()*.

Mock is very easy to use and is designed for use with *unittest*. Mock is based on the 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks.

There is a backport of *unittest.mock* for earlier versions of Python, available as mock on PyPI.

27.5.1 Quick Guide

Mock and *MagicMock* objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used:

```
>>> 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` allows you to perform side effects, including raising an exception when a mock is called:

```
>>> 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)
```

Mock has many other ways you can configure it and control its behaviour. For example the *spec* argument configures the mock to take its specification from another object. Attempting to access attributes or methods on the mock that don't exist on the spec will fail with an *AttributeError*.

The *patch()* decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> 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: When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `module.ClassName1` is passed in first.

With *patch()* it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

As well as a decorator *patch()* can be used as a context manager in a with statement:

```
>>> 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)
```

There is also *patch.dict()* for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock supports the mocking of Python *magic methods*. The easiest way of using magic methods is with the *MagicMock* class. It allows you to do things like:

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock allows you to assign functions (or other Mock instances) to magic methods and they will be called appropriately. The *MagicMock* class is just a Mock variant that has all of the magic methods pre-created for you (well, all the useful ones anyway).

The following is an example of using magic methods with the ordinary Mock class:

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

For ensuring that the mock objects in your tests have the same api as the objects they are replacing, you can use *auto-speccing*. Auto-speccing can be done through the *autospec* argument to patch, or the *create_autospec()* function. Auto-speccing creates mock objects that have the same attributes and methods as the objects they are replacing, and any functions and methods (including constructors) have the same call signature as the real object.

This ensures that your mocks will fail in the same way as your production code if they are used incorrectly:

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> mock_function = create_autospec(function, return_value='fishy')
>>> 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: <lambda>() takes exactly 3 arguments (1 given)
```

create_autospec() can also be used on classes, where it copies the signature of the __init__ method, and on callable objects where it copies the signature of the __call__ method.

27.5.2 The Mock Class

Mock is a flexible mock object intended to replace the use of stubs and test doubles throughout your code. Mocks are callable and create attributes as new mocks when you access them[1]. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

MagicMock is a subclass of *Mock* with all the magic methods pre-created and ready to use. There are also non-callable variants, useful when you are mocking out objects that aren't callable: *NonCallableMock* and *NonCallableMagicMock*

The *patch()* decorators makes it easy to temporarily replace classes in a particular module with a *Mock* object. By default *patch()* will create a *MagicMock* for you. You can specify an alternative class of *Mock* using the *new_callable* argument to *patch()*.

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *__class__* returns the class of the spec object. This allows mocks to pass *isinstance()* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

  Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable.

  A *side_effect* can be cleared by setting it to None.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.

[1] The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead raises an *AttributeError*. This is because the interpreter will often implicitly request these methods, and gets *very* confused to get a new Mock object when it expects a magic method. If you need magic method support see *magic methods*.

- *unsafe*: By default if any attribute starts with *assert* or *assret* will raise an *AttributeError*. Passing `unsafe=True` will allow access to these attributes.

  버전 3.5에 추가.

- *wraps*: Item for the mock object to wrap. If *wraps* is not `None` then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

  If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.

- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created. See the *configure_mock()* method for details.

assert_called()
Assert that the mock was called at least once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

버전 3.6에 추가.

assert_called_once()
Assert that the mock was called exactly once.

```
>>> 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.
```

버전 3.6에 추가.

assert_called_with(**args*, ***kwargs*)
This method is a convenient way of asserting that calls are made in a particular way:

```
>>> 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*)
Assert that the mock was called exactly once and that that call was with the specified arguments.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call(**args*, ***kwargs*)
assert the mock has been called with the specified arguments.

The assert passes if the mock has *ever* been called, unlike *assert_called_with()* and *assert_called_once_with()* that only pass if the call is the most recent one, and in the case of *assert_called_once_with()* it must also be the only call.

```
>>> 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*)
assert the mock has been called with the specified calls. The *mock_calls* list is checked for the calls.

If *any_order* is false then the calls must be sequential. There can be extra calls before or after the specified calls.

If *any_order* is true then the calls can be in any order, but they must all appear in *mock_calls*.

```
>>> 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()
Assert the mock was never called.

```
>>> 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.
```

버전 3.5에 추가.

reset_mock(***, *return_value=False*, *side_effect=False*)
The reset_mock method resets all the call attributes on a mock object:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

버전 3.6에서 변경: Added two keyword only argument to the reset_mock function.

This can be useful where you want to make a series of assertions that reuse the same object. Note that *reset_mock()* *doesn't* clear the return value, *side_effect* or any child attributes you have set using normal assignment by default. In case you want to reset *return_value* or *side_effect*, then pass the corresponding parameter as `True`. Child mocks and the return value mock (if any) are reset as well.

참고: *return_value*, and *side_effect* are keyword only argument.

---

mock_add_spec(*spec*, *spec_set=False*)

Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is true then only attributes on the spec can be set.

attach_mock(*mock*, *attribute*)

Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the *method_calls* and *mock_calls* attributes of this one.

configure_mock(***kwargs*)

Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
>>> 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
```

The same thing can be achieved in the constructor call to mocks:

```
>>> 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
```

configure_mock() exists to make it easier to do configuration after the mock has been created.

__dir__()

Mock objects limit the results of `dir(some_mock)` to useful results. For mocks with a *spec* this includes all the permitted attributes for the mock.

See *FILTER_DIR* for what this filtering does, and how to switch it off.

_get_child_mock(***kw*)

Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

called

A boolean representing whether or not the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

An integer telling you how many times the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

Set this to configure the value returned by calling the mock:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure it in the normal way:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

This can either be a function to be called when the mock is called, an iterable or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the *DEFAULT* singleton the call to the mock will then return whatever the function returns. If the function returns *DEFAULT* then the mock will return its normal value (from the *return_value*).

If you pass in an iterable, it is used to retrieve an iterator which must yield a value on every call. This value can either be an exception instance to be raised, or a value to be returned from the call to the mock (*DEFAULT* handling is identical to the function case).

An example of a mock that raises an exception (to test exception handling of an API):

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Using *side_effect* to return a sequence of values:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Using a callable:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect can be set in the constructor. Here's an example that adds one to the value the mock is called with and returns it:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

Setting *side_effect* to `None` clears it:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
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 is any ordered arguments the mock was called with (or an empty tuple) and the second member is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
```

call_args, along with members of the lists *call_args_list*, *method_calls* and *mock_calls* are *call* objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See *calls as tuples*.

call_args_list

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The *call* object can be used for conveniently constructing lists of calls to compare with *call_args_list*.

```
>>> 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!'},)]
>>> mock.call_args_list == expected
True
```

Members of *call_args_list* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

method_calls

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes:

```
>>> 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()]
```

Members of *method_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

mock_calls

mock_calls records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
<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
```

Members of *mock_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

참고: The way *mock_calls* are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal:

```
>>> 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__

Normally the *__class__* attribute of an object will return its type. For a mock object with a `spec`, `__class__` returns the spec class instead. This allows mock objects to pass *isinstance()* tests for the object they are replacing / masquerading as:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ is assignable to, this allows a mock to pass an *isinstance()* check without forcing you to use a spec:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

A non-callable version of *Mock*. The constructor parameters have the same meaning of *Mock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a `spec` or `spec_set` are able to pass *isinstance()* tests:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The *Mock* classes have support for mocking magic methods. See *magic methods* for the full details.

The mock classes and the *patch()* decorators all take arbitrary keyword arguments for configuration. For the *patch()* decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can't use dotted names directly in a call you have to create a dictionary and unpack it using **:

```
>>> 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
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name:

```
>>> 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)
```

This applies to *assert_called_with()*, *assert_called_once_with()*, *assert_has_calls()* and *assert_any_call()*. When *Autospeccing*, it will also apply to method calls on the mock object.

버전 3.4에서 변경: Added signature introspection on specced and autospecced mock objects.

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

A mock intended to be used as a property, or other descriptor, on a class. *PropertyMock* provides __get__() and __set__() methods so you can specify a return value when it is fetched.

Fetching a *PropertyMock* instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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)]
```

Because of the way mock attributes are stored you can't directly attach a *PropertyMock* to a mock object. Instead you can attach it to the mock type object:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

Calling

Mock objects are callable. The call will return the value set as the *return_value* attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like *call_args* and *call_args_list*.

If *side_effect* is set then it will be called after the call has been recorded, so if `side_effect` raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make *side_effect* an exception class or instance:

```
>>> 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')]
```

If `side_effect` is a function then whatever that function returns is what calls to the mock return. The `side_effect` function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```
>>> def side_effect(value):
...     return value + 1
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return `mock.return_value` from inside `side_effect`, or return *DEFAULT*:

```
>>> 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
```

To remove a `side_effect`, and return to the default behaviour, set the `side_effect` to `None`:

```
>>> 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
```

The `side_effect` can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a *StopIteration* is raised):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a *`hasattr()`* call, or raise an *`AttributeError`* when an attribute is fetched. You can do this by providing an object as a `spec` for a mock, but that isn't always convenient.

You "block" attributes by deleting them. Once deleted, accessing an attribute will raise an *`AttributeError`*.

```
>>> 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
```

Mock names and the name attribute

Since "name" is an argument to the *`Mock`* constructor, if you want your mock object to have a "name" attribute you can't just pass it in at creation time. There are two alternatives. One option is to use *`configure_mock()`*:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the "name" attribute after mock creation:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a "child" of that mock. Calls to the child are recorded in the *method_calls* and *mock_calls* attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the "parenting" if for some reason you don't want it to happen.

```
>>> 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
[]
```

Mocks created for you by *patch()* are automatically given names. To attach mocks that have names to a parent you use the *attach_mock()* method:

```
>>> 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')]
```

27.5.3 The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

참고: `patch()` is straightforward to use. The key is to do the patching in the right namespace. See the section *where to patch*.

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

`patch()` acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with a `MagicMock`. If `patch()` is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If `patch()` is used as a context manager the created mock is returned by the context manager.

target should be a string in the form `'package.module.ClassName'`. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling `patch()` from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the `MagicMock` if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default `MagicMock` is used.

A more powerful form of *spec* is *autospec*. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a `TypeError` if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class. See the `create_autospec()` function and *Autospeccing*.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default `patch()` will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again after the patched function has exited. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

참고: 버전 3.5에서 변경: If you are patching builtins in a module then you don't need to pass `create=True`, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. `patch()` finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `'test'`, which matches the way `unittest` finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if `patch()` is creating a mock object for you.

`patch()` takes arbitrary keyword arguments. These will be passed to the `Mock` (or *new_callable*) on construction.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

patch() as function decorator, creating the mock for you and passing it into the decorated function:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a *MagicMock* *instance*. If the class is instantiated in the code under test then it will be the *return_value* of the mock that will be used.

If the class is instantiated multiple times you could use *side_effect* to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the `return_value`. For example:

```
>>> 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'
...
```

If you use *spec* or *spec_set* and *patch()* is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default *MagicMock* for the created mock. For example, if you wanted a *NonCallableMock* to be used:

```
>>> 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
```

Another use case might be to replace an object with an *io.StringIO* instance:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When *patch()* is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the *return_value* and *side_effect*, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a *patch()* call using **:

```
>>> 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
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with *AttributeError*:

```
>>> @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'
```

but adding `create=True` in the call to *patch()* will make the previous example work as expected:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

patch.object

patch.**object**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch the named member (*attribute*) on an object (*target*) with a mock object.

patch.object() can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *spec_set*, *autospec* and *new_callable* have the same meaning as for *patch()*. Like *patch()*, *patch.object()* takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator *patch.object()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call *patch.object()* with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

spec, *create* and the other arguments to *patch.object()* have the same meaning as they do for *patch()*.

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.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of `(key, value)` pairs.

If *clear* is true then the dictionary will be cleared before the new values are set.

patch.dict() can also be called with arbitrary keyword arguments to set values in the dictionary.

patch.dict() can be used as a context manager, decorator or class decorator. When used as a class decorator *patch.dict()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

patch.dict() can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the *patch.dict()* call to set values in the dictionary:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either `__iter__()` or `__contains__()`.

```
>>> 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*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

Use *DEFAULT* as the value if you want *patch.multiple()* to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when *patch.multiple()* is used as a context manager.

patch.multiple() can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *autospec* and *new_callable* have the same meaning as for *patch()*. These arguments will be applied to *all* patches done by *patch.multiple()*.

When used as a class decorator *patch.multiple()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

If you want *patch.multiple()* to create mocks for you, then you can use *DEFAULT* as the value. If you use *patch.multiple()* as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> 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() can be nested with other `patch` decorators, but put arguments passed by keyword *after* any of the standard arguments created by *patch()*:

```
>>> @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()
```

If *patch.multiple()* is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name:

```
>>> 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
...
```

patch methods: start and stop

All the patchers have `start()` and `stop()` methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call *patch()*, *patch.object()* or *patch.dict()* as normal and keep a reference to the returned `patcher` object. You can then call `start()` to put the patch in place and `stop()` to undo it.

If you are using *patch()* to create a mock for you then it will be returned by the call to `patcher.start`.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the `setUp` method of a `TestCase`:

```
>>> class MyTest(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: If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. *`unittest.TestCase.addCleanup()`* makes this easier:

```
>>> class MyTest(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
...
```

As an added bonus you no longer need to keep a reference to the `patcher` object.

It is also possible to stop all patches which have been started by using *`patch.stopall()`*.

`patch.`**`stopall`**`()`

Stop all active patches. Only stops patches started with `start`.

patch builtins

You can patch any builtins within a module. The following example patches builtin *ord()*:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with `'test'` as being test methods. This is the same way that the *unittest.TestLoader* finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`:

```
>>> 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
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```
>>> @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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

patch() works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using *patch()*. The problem is that when we import module b, which we will have to do then it imports `SomeClass` from module a. If we use *patch()* to mock out `a.SomeClass` then it will have no effect on our test; module b already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module b, where we have imported it. The patching should look like:

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a import SomeClass` module b does `import a` and `some_function` uses `a.SomeClass`. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch `a.SomeClass` instead:

```
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both *patch* and *patch.object* correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object.

27.5.4 MagicMock and magic method support

Mocking Magic Methods

Mock supports mocking the Python protocol methods, also known as "magic methods". This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods[2], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument[3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> 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)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement:

```
>>> 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)
```

Calls to magic methods do not appear in *method_calls*, but they are recorded in *mock_calls*.

참고: If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an *AttributeError*.

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`

[2] Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

[3] The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

- __dir__, __format__ and __subclasses__
- __floor__, __trunc__ and __ceil__
- Comparisons: __lt__, __gt__, __le__, __ge__, __eq__ and __ne__
- Container methods: __getitem__, __setitem__, __delitem__, __contains__, __len__, __iter__, __reversed__ and __missing__
- Context manager: __enter__ and __exit__
- Unary numeric methods: __neg__, __pos__ and __invert__
- The numeric methods (including right hand and in-place variants): __add__, __sub__, __mul__, __matmul__, __div__, __truediv__, __floordiv__, __mod__, __divmod__, __lshift__, __rshift__, __and__, __xor__, __or__, and __pow__
- Numeric conversion methods: __complex__, __int__, __float__ and __index__
- Descriptor methods: __get__, __set__ and __delete__
- Pickling: __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ and __setstate__

The following methods exist but are *not* supported as they are either in use by mock, can't be set dynamically, or can cause problems:

- __getattr__, __setattr__, __init__ and __new__
- __prepare__, __instancecheck__, __subclasscheck__, __del__

Magic Mock

There are two MagicMock variants: *MagicMock* and *NonCallableMagicMock*.

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock is a subclass of *Mock* with default implementations of most of the magic methods. You can use MagicMock without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for *Mock*.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

A non-callable version of *MagicMock*.

The constructor parameters have the same meaning as for *MagicMock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

The magic methods are setup with *MagicMock* objects, so you can configure them and use them in the usual way:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren't interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults:

- __lt__: NotImplemented
- __gt__: NotImplemented
- __le__: NotImplemented
- __ge__: NotImplemented
- __int__: 1
- __contains__: False
- __len__: 0
- __iter__: iter([])
- __exit__: False
- __complex__: 1j
- __float__: 1.0
- __bool__: True
- __index__: 1
- __hash__: default hash for the mock
- __str__: default str for the mock
- __sizeof__: default sizeof for the mock

For example:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality methods, __eq__() and __ne__(), are special. They do the default equality comparison on identity, using the *side_effect* attribute, unless you change their return value to return something else:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

The return value of MagicMock.__iter__() can be any iterable object and isn't required to be an iterator:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value *is* an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` and `__delete__`
- `__reversed__` and `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- `__getformat__` and `__setformat__`

27.5.5 Helpers

sentinel

`unittest.mock.`**`sentinel`**

The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

버전 3.7에서 변경: The `sentinel` attributes now preserve their identity when they are *copied* or *pickled*.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. *sentinel* provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch `method` to return `sentinel.some_object`:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**

The *DEFAULT* object is a pre-created sentinel (actually sentinel.DEFAULT). It can be used by *side_effect* functions to indicate that the normal return value should be used.

call

unittest.mock.**call**(**args*, ***kwargs*)

call() is a helper object for making simpler assertions, for comparing with *call_args*, *call_args_list*, *mock_calls* and *method_calls*. *call()* can also be used with *assert_has_calls()*.

```
>>> 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**()

For a call object that represents multiple calls, *call_list()* returns a list of all the intermediate calls as well as the final call.

call_list is particularly useful for making assertions on "chained calls". A chained call is multiple calls on a single line of code. This results in multiple entries in *mock_calls* on a mock. Manually constructing the sequence of calls can be tedious.

call_list() can construct the sequence of calls from the same chained call:

```
>>> 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
```

A call object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn't particularly interesting, but the call objects that are in the *Mock.call_args*, *Mock.call_args_list* and *Mock.mock_calls* attributes can be introspected to get at the individual arguments they contain.

The call objects in *Mock.call_args* and *Mock.call_args_list* are two-tuples of (positional args, keyword args) whereas the call objects in *Mock.mock_calls*, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their "tupleness" to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> args, kwargs = kall
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> 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
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with the correct signature.

If *spec_set* is `True` then attempting to set attributes that don't exist on the spec object will raise an *AttributeError*.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing `instance=True`. The returned mock will only be callable if instances of the mock are callable.

create_autospec() also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See *Autospeccing* for examples of how to use auto-speccing with *create_autospec()* and the *autospec* argument to *patch()*.

ANY

unittest.mock.**ANY**

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of *call_args* and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to *assert_called_with()* and *assert_called_once_with()* will then succeed no matter what was passed in.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY can also be used in comparisons with call lists like *mock_calls*:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR is a module level variable that controls the way mock objects respond to *dir()* (only for Python 2.6 or more recent). The default is `True`, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set `mock.FILTER_DIR = False`.

With filtering on, `dir(some_mock)` shows only useful attributes and will include any dynamically created attributes that wouldn't normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven't been accessed yet:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

Many of the not-very-useful (private to *Mock* rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling *dir()* on a *Mock*. If you dislike this behaviour you can switch it off by setting the module level switch *FILTER_DIR*:

```
>>> 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__',
 ...
```

Alternatively you can just use `vars(my_mock)` (instance members) and `dir(type(my_mock))` (type members) to bypass the filtering irrespective of `mock.FILTER_DIR`.

mock_open

`unittest.mock.`**`mock_open`**(*mock=None*, *read_data=None*)

A helper function to create a mock to replace the use of *open()*. It works for *open()* called directly or used as a context manager.

The *mock* argument is the mock object to configure. If `None` (the default) then a *MagicMock* will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the `read()`, *readline()*, and *readlines()* methods of the file handle to return. Calls to those methods will take data from *read_data* until it is depleted. The mock of these methods is pretty simplistic: every time the *mock* is called, the *read_data* is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI can offer a realistic filesystem for testing.

버전 3.4에서 변경: Added *readline()* and *readlines()* support. The mock of `read()` changed to consume *read_data* rather than returning it on each call.

버전 3.5에서 변경: *read_data* is now reset on each call to the *mock*.

버전 3.7.1에서 변경: Added `__iter__()` to implementation so that iteration (such as in for loops) correctly consumes *read_data*.

Using *open()* as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to *open()* it is the *returned object* that is used as a context manager (and has `__enter__()` and `__exit__()` called).

Mocking context managers with a *MagicMock* is common enough and fiddly enough that a helper function is useful.

```
>>> 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__(),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 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')
```

And for reading files:

```
>>> 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'
```

Autospeccing

Autospeccing is based on the existing `spec` feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a *TypeError* if they are called incorrectly.

Before I explain how auto-speccing works, here's why it is needed.

Mock is a very powerful and flexible object, but it suffers from two flaws when used to mock out objects from a system under test. One of these flaws is specific to the *Mock* api and the other is a more general problem with using mock objects.

First the problem specific to *Mock*. *Mock* has two assert methods that are extremely handy: *assert_called_with()* and *assert_called_once_with()*.

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

Because mocks auto-create attributes on demand, and allow you to call them with arbitrary arguments, if you misspell one of these assert methods then your assertion is gone:

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

Your tests can pass silently and incorrectly because of the typo.

The second issue is more general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don't test how your units are "wired together" there is still lots of room for bugs that tests might have caught.

mock already provides a feature to help with this, called speccing. If you use a class or instance as the spec for a mock then you can only access attributes on the mock that exist on the real class:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()
```

Auto-speccing solves this problem. You can either pass autospec=True to *patch()* / *patch.object()* or use the *create_autospec()* function to create a mock with a spec. If you use the autospec=True argument to *patch()* then the object that is being replaced will be used as the spec object. Because the speccing is done "lazily" (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here's an example of it in use:

```
>>> 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='...'>
```

You can see that request.Request has a spec. request.Request takes two arguments in the constructor (one of which is *self*). Here's what happens if we try to call it incorrectly:

```
>>> req = request.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

Request objects are not callable, so the return value of instantiating our mocked out request.Request is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add `autospec=True` to your existing *patch()* calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through *patch()* there is a *create_autospec()* for creating autospecced mocks directly:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe[4].

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. *autospec* can't know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> 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'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario:

```
>>> 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'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in `__init__()` then providing them via class attributes (shared between instances of course) is faster too. e.g.

[4] This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to *dir()* - that are done.

```
class Something:
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn't let you access *any* attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, autospec doesn't use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks):

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn't to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully *patch()* supports this - you can simply pass the alternative object as the *autospec* argument:

```
>>> 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='...'>
```

Sealing mocks

unittest.mock.**seal**(*mock*)

Seal will disable the automatic creation of mocks when accessing an attribute of the mock being sealed or any of its attributes that are already mocks recursively.

If a mock instance with a name or a spec is assigned to an attribute it won't be considered in the sealing chain. This allows one to prevent seal from fixing part of the mock object.

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

버전 3.7에 추가.

27.6 unittest.mock — getting started

버전 3.3에 추가.

27.6.1 Using Mock

Mock Patching Methods

Common uses for *Mock* objects include:

- Patching methods
- Recording method calls on objects

You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Once our mock has been used (`real.method` in this example) it has methods and attributes that allow you to make assertions about how it has been used.

참고: In most of these examples the *Mock* and *MagicMock* classes are interchangeable. As the `MagicMock` is the more capable class it makes a sensible one to use by default.

Once the mock has been called its *called* attribute is set to `True`. More importantly we can use the *assert_called_with()* or *assert_called_once_with()* method to check that it was called with the correct arguments.

This example tests that calling `ProductionClass().method` results in a call to the `something` method:

```
>>> 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)
```

Mock for Method Calls on an Object

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple `ProductionClass` below has a `closer` method. If it is called with an object then it calls `close` on it.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

So to test it we need to pass in an object with a `close` method and check that it was called correctly.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

We don't have to do any work to provide the 'close' method on our mock. Accessing close creates it. So, if 'close' hasn't already been called then accessing it in the test will create it, but *`assert_called_with()`* will raise a failure exception.

Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by *calling the class*. This means you access the "mock instance" by looking at the return value of the mocked class.

In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to *`patch()`* replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock *`return_value`*.

```
>>> def some_function():
...     instance = module.Foo()
...     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'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The *mock_calls* attribute records all calls to child attributes of the mock - and also to their children.

```
>>> 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)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the *call* object to construct lists for comparing with `mock_calls`:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use *call* to construct the set of calls in a "chained call" like this for easy assertion afterwards:

```
>>> 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
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is *side_effect*. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```
>>> 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
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

Mock allows you to provide an object as a specification for the mock, using the *spec* keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: object has no attribute 'old_method'
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments:

```
>>> 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)
```

If you want this smarter matching to also work with method calls on the mock, you can use *auto-speccing*.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use *spec_set* instead of *spec*.

27.6.2 Patch Decorators

참고: With *patch()* it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: *patch()*, *patch.object()* and *patch.dict()*. patch takes a single string, of the form package.module.Class.attribute to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. 'patch.object' takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

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()
```

If you are patching a module (including *builtins*) then use *patch()* instead of *patch.object()*:

```
>>> 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"
```

The module name can be 'dotted', in the form package.module if needed:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

A nice pattern is to actually decorate test methods themselves:

```
>>> 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
```

If you want to patch with a Mock, you can use *patch()* with only one argument (or *patch.object()* with two arguments). The mock will be created for you and passed into the test function / method:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern:

```
>>> 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()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

There is also *`patch.dict()`* for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use *`patch()`* to create a mock for you, you can get a reference to the mock using the "as" form of the with statement:

```
>>> 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)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with "test".

27.6.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the *return_value* attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new *Mock* is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place…

So, suppose we have some code that looks a little bit like this:

```
>>> 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
```

Assuming that `BackendProvider` is already well tested, how do we test `method()`? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is 'file-like', so we'll ensure that our response object uses the builtin *open()* as its `spec`.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

We can do that in a slightly nicer way using the *configure_mock()* method to directly set the return value for us:

```
>>> 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)
```

With these we monkey patch the "mock backend" in place and can make the real call:

```
>>> something.backend = mock_backend
>>> something.method()
```

Using *mock_calls* we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use *call.call_list()* to create this list of calls for us:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to *datetime.date.today()* to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately *datetime.date* is written in C, and so I couldn't just monkey-patch out the static `date.today()` method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The *patch decorator* is used here to mock out the `date` class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> 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)
...
```

Note that we don't patch *datetime.date* globally, we patch `date` in the module that *uses* it. See *where to patch*.

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and friends) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry.

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over[1].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is *__iter__()*, so we can mock this using a *MagicMock*.

Here's an example class with an "iter" method implemented as a generator:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
```

(다음 페이지에 계속)

[1] There are also generator expressions and more advanced uses of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is: Generator Tricks for Systems Programmers.

(이전 페이지에서 계속)

```
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its "iter" method?

To configure the values returned from the iteration (implicit in the call to *list*), we need to configure the object returned by the call to `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use *patch()* (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(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'
```

An alternative way of managing patches is to use the *patch methods: start and stop*. These allow you to move the patching into your `setUp` and `tearDown` methods.

```
>>> class MyTest(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()
```

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. *unittest.TestCase.addCleanup()* makes this easier:

```
>>> class MyTest(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()
```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get self passed in. The workaround is to patch the unbound method with a real function instead. The *patch()* decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted:

```
>>> 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)
```

If we don't use `autospec=True` then the unbound method is patched out with a Mock instance instead, and isn't called with `self`.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the `assert_called_once_with()` method that also asserts that the `call_count` is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected to be called once. Called 2 times.
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use *`call_args_list`*:

```
>>> 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()]
```

The *`call`* helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in 'mymodule':

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```
>>> 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(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the `side_effect` functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> 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` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

참고: If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

An alternative approach is to create a subclass of *Mock* or *MagicMock* that copies (using *copy.deepcopy()*) the arguments. Here's an example implementation:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super(CopyingMock, self).__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```
>>> class MyTest(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
```

With unittest `cleanup` functions and the *patch methods: start and stop* we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     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
```

Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with *MagicMock*, which will behave like a dictionary, and using *side_effect* to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like *call_args_list* to assert about how the dictionary was used:

```
>>> 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: An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the *spec* (or *spec_set*) argument so that the `MagicMock` created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a *KeyError* if you try to access a key that doesn't exist.

```
>>> 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'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> 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, 'c': 3, 'b': 'fish', 'd': 'eggs'}
```

Mock subclasses and their attributes

There are various reasons why you might want to subclass *Mock*. One reason might be to add helper methods. Here's a silly example:

```
>>> 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
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`[2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
```

(다음 페이지에 계속)

[2] An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

(이전 페이지에서 계속)

```
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

`Mock` (in all its flavours) uses a method called `_get_child_mock` to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor:

```
>>> 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)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent "up front costs" by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the *sys.modules* dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in *sys.modules*, so usually when you import something you get a module back. This need not be the case however.

This means you can use *patch.dict()* to *temporarily* put a mock in place in *sys.modules*. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> 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()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in *sys.modules*.

This also works for the `from module import name` form:

```
>>> 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()
```

With slightly more work you can also mock package imports:

```
>>> 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()
```

Tracking order of calls and less verbose call assertions

The *Mock* class allows you to track the *order* of method calls on your mock objects through the *method_calls* attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *mock_calls* to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```
>>> 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()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the *attach_mock()* method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         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()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the *assert_has_calls()* method. This takes a list of calls (constructed with the *call* object). If that sequence of calls are in *mock_calls* then the assert succeeds.

```
>>> 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)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```
>>> 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)
```

More complex argument matching

Using the same basic concept as *ANY* we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use *assert_called_with()* we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a 'standard' call to `assert_called_with` isn't sufficient:

```
>>> 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))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Traceback (most recent call last):
    ...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```
>>> 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
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> 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)
...
```

Putting all this together:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an *`AssertionError`* is raised:

```
>>> 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...>,), {})
```

With a bit of tweaking you could have the comparison function raise the *`AssertionError`* directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality).

27.7 2to3 - 파이썬 2에서 파이썬 3으로 자동 코드 변환

2to3는 파이썬 2.x 소스 코드를 유효한 파이썬 3.x 코드로 변환하기 위해 일련의 변환자*(fixers)*를 적용하는 프로그램입니다. 표준 라이브러리는 많은 양의 변환자를 제공하고 있어 코드 대부분을 처리할 수 있을 것입니다. 2to3에서 사용하는 모듈인 *lib2to3* 는 유연하고 제네릭합니다. 따라서 2to3 프로그램을 위해 당신만의 변환자를 작성할 수 있습니다. 또한 *lib2to3* 는 파이썬 코드를 자동으로 수정해주는 커스텀 응용 프로그램에서도 사용할 수 있습니다.

27.7.1 2to3 사용법

파이썬 인터프리터가 설치될 때, 보통 2to3 스크립트도 같이 설치됩니다. 2to3 스크립트 파일은 파이썬 루트 디렉터리의 하위 디렉터리인 `Tools/scripts` 에서 찾을 수 있습니다.

2to3의 기본 인자는 변환하고자 하는 파일이나 디렉터리 리스트입니다. 디렉터리의 경우 하위 폴더의 파이썬 소스까지 적용됩니다.

샘플 파이썬 2.x 코드가 여기 있습니다. `example.py`:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

명령줄에서 2to3를 실행하면 이 코드를 파이썬 3.x 코드로 바꿀 수 있습니다:

```
$ 2to3 example.py
```

원본 파일과 변환 결과를 비교한 차이점(diff)이 출력됩니다. 2to3은 원본 소스 파일에 필요한 수정사항을 바로 적용할 수도 있습니다. (`-n` 옵션이 적용되지 않았다면 원본 파일에 대한 백업이 생성될 것입니다.) `-w` 옵션을 사용하면 바로 원본 파일이 수정됩니다.

```
$ 2to3 -w example.py
```

`example.py` 를 변환한 결과는 다음과 같습니다.

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

변환 과정에서 들여쓰기와 주석은 그대로 보존됩니다.

기본적으로 2to3는 미리 정의된 변환자 를 사용하여 실행됩니다. `-l` 옵션을 사용하면 사용 가능한 모든 변환자를 볼 수 있습니다. 특정 변환자만 명시적으로 설정하고 싶으시면 `-f` 를 사용하시면 됩니다. 마찬가지로 `-x` 옵션으로 특정 변환자를 비활성화할 수도 있습니다. 다음 예는 `imports` 와 `has_key` 변환자만 사용한 것입니다.

```
$ 2to3 -f imports -f has_key example.py
```

다음은 `apply` 변환자만 빼고 모든 변환자를 실행하는 명령어입니다.

```
$ 2to3 -x apply example.py
```

몇몇 변환자는 기본적으로 실행되지 않기 때문에 명시적으로 명령줄에서 설정해야 합니다. 기본 변환자에 `idioms` 변환자를 추가한 예시가 여기 있습니다.

```
$ 2to3 -f all -f idioms example.py
```

모든 기본 변환자를 활성화하기 위해 `all` 값을 사용한 것을 주목해주세요.

때때로 2to3는 자동 변환을 하지 못하고 당신의 코드에서 수정이 필요한 부분을 찾을 수도 있습니다. 이러면 파일의 비교 결과 아래에 경고 문구를 출력할 것입니다. 당신은 이 소스코드를 3.x 버전에 맞도록 경고 사항을 수정해야 합니다.

2to3는 doctest도 수정할 수 있습니다. 이것을 활성화하기 위해서는 `-d` 옵션을 사용하세요. 이것은 오직 doctest만 수정한다는 것을 명심하세요. 이것을 사용하기 위해서 꼭 적합한 파이썬 모듈이 필요한 것은 아닙니다. 예를 들어 reST 문서에 있는 예와 같은 doctest도 이 옵션과 함께 수정할 수 있습니다.

`-v` 옵션은 변환 과정 동안 더 자세한 정보를 출력하게 해줍니다.

어떤 print 문장의 경우는 문장 또는 함수 호출로 파싱될 수 있기 때문에 2to3이 print 함수를 포함한 파일을 항상 처리할 수 있는 것은 아닙니다. 2to3이 `from __future__ import print_function` 이란 컴파일러 지시어를 찾았다면 2to3는 *`print()`* 를 함수로 처리하도록 내부 처리 문법을 변경할 것입니다. 이러한 변경은 `-p` 옵션을 가지고 직접 활성화할 수도 있습니다. 출력 문장이 이미 변경된 코드에 변환자를 실행하기 위해 `-p` 옵션을 사용하세요.

`-o` 또는 `--output-dir` 옵션을 사용하면 출력 파일이 쓰일 디렉터리를 설정할 수 있습니다.:option:*!-n* 옵션은 입력 파일을 덮어쓰지 않아 백업 파일이 필요 없을 때 사용할 수 있습니다.

버전 3.2.3에 추가: `-o` 옵션이 추가되었습니다.

`-W` 또는 `--write-unchanged-files` 옵션은 변경 사항이 없더라도 항상 출력 파일을 쓰도록 합니다. 이것을 `-o` 옵션과 함께 쓰면 한 디렉터리에 있는 전체 파이썬 소스 트리를 파이썬 3.x로 변환해서 다른 디렉터리로 복사할 때 유용하게 사용할 수 있습니다. 이치에 맞게 하기 위해 이 옵션은 `-w` 옵션을 포함하고 있습니다.

버전 3.2.3에 추가: `-W` 옵션이 추가되었습니다.

`--add-suffix` 옵션은 모든 출력 파일 이름 뒤에 추가할 문자열을 지정합니다. 다른 파일 이름으로 저장할 때 백업 파일이 필요하지 않다면 `-n` 옵션을 같이 사용해야 합니다. 예시:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

이 명령어는 출력 파일의 이름을 `example.py3` 로 만들어 줍니다.

버전 3.2.3에 추가: `--add-suffix` 옵션이 추가되었습니다.

한 디렉터리에서 다른 디렉터리로 전체 프로젝트를 변환하고 싶을 때는 다음과 같이 하면 됩니다.

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

27.7.2 변환자 목록

변환되는 코드의 각각 단계는 변환자 안에 캡슐화되어 있습니다. `2to3 -l` 명령어를 실행하면 변환자 목록을 보실 수 있습니다. 위에 적어놓은 것 과 같이, 각 변환자는 개별적으로 활성화/비활성화를 할 수 있습니다. 변환자들에 대한 자세한 설명이 아래에 있습니다.

apply
: `apply()` 사용을 제거합니다. 예를 들어 `apply(function, *args, **kwargs)` 를 `function(*args, **kwargs)` 로 변경합니다.

asserts
: 폐지된 *`unittest`* 메서드 이름을 올바른 것으로 변경합니다.

변경 전	변경 후
failUnlessEqual(a, b)	*assertEqual(a, b)*
assertEquals(a, b)	*assertEqual(a, b)*
failIfEqual(a, b)	*assertNotEqual(a, b)*
assertNotEquals(a, b)	*assertNotEqual(a, b)*
failUnless(a)	*assertTrue(a)*
assert_(a)	*assertTrue(a)*
failIf(a)	*assertFalse(a)*
failUnlessRaises(exc, cal)	*assertRaises(exc, cal)*
failUnlessAlmostEqual(a, b)	*assertAlmostEqual(a, b)*
assertAlmostEquals(a, b)	*assertAlmostEqual(a, b)*
failIfAlmostEqual(a, b)	*assertNotAlmostEqual(a, b)*
assertNotAlmostEquals(a, b)	*assertNotAlmostEqual(a, b)*

basestring
basestring 을 *str* 로 변환합니다.

buffer
buffer 를 *memoryview* 로 변환합니다. *memoryview* API가 buffer API와 비슷하긴 하지만 완전히 같진 않아서 이 변환자는 선택적으로 실행됩니다.

dict
딕셔너리 이터레이션 메서드를 변환합니다. dict.iteritems() 를 *dict.items()* 로, dict.iterkeys() 를 *dict.keys()* 로, dict.itervalues() 를 *dict.values()* 로 변경합니다. 마찬가지로 dict.viewitems(), dict.viewkeys(), dict.viewvalues() 를 각각 *dict.items()*, *dict.keys()*, *dict.values()* 로 변경합니다. 기존의 *dict.items()*, *dict.keys()*, *dict.values()* 의 사용을 *list* 로 감싸도록 바꿉니다.

except
except X, T 를 except X as T 로 변환합니다.

exec
exec 문장을 *exec()* 함수로 변환합니다.

execfile
execfile() 사용을 제거합니다. execfile() 에 사용되는 인자는 *open()*, *compile()*, *exec()* 을 사용하도록 바뀝니다.

exitfunc
sys.exitfunc 대입이 *atexit* 모듈을 사용하도록 바뀝니다.

filter
filter() 함수 사용을 *list* 로 감싸도록 바꿉니다.

funcattrs
이름이 변경된 함수 어트리뷰트를 변환합니다. 예를 들어 my_function.func_closure 를 my_function.__closure__ 로 변경합니다.

future
from __future__ import new_feature 구문을 제거합니다.

getcwdu
os.getcwdu() 를 *os.getcwd()* 로 변경합니다.

has_key
dict.has_key(key) 를 key in dict 로 바꿉니다.

idioms
이 선택적인 변환자는 이디엄을 더 사용하도록 파이썬 코드를 변환해줍니다. type(x) is SomeClass

나 `type(x) == SomeClass` 같은 형 비교는 `isinstance(x, SomeClass)` 로 변환합니다. `while 1` 는 `while True` 로 변환합니다. 또한 이 변환자는 *sorted()* 가 올바른 위치에 사용될 수 있도록 수정합니다. 예를 들어 다음 코드는

```
L = list(some_iterable)
L.sort()
```

아래와 같이 변경됩니다.

```
L = sorted(some_iterable)
```

import
같은 단계 경로의 임포트를 찾아 상대 경로 임포트로 변경합니다.

imports
표준 라이브러리에 있는 모듈의 이름 변경 사항을 처리합니다.

imports2
표준 라이브러리에 있는 또 다른 모듈의 이름 변경 사항을 처리합니다. 기술적인 제한 사항 때문에 *imports* 변환자와 분리했습니다.

input
`input(prompt)` 를 `eval(input(prompt))` 로 변경합니다.

intern
`intern()` 를 *sys.intern()* 로 변경합니다.

isinstance
isinstance() 의 두 번째 인자에서 중복된 형을 수정합니다. 예를 들어 `isinstance(x, (int, int))` 를 `isinstance(x, int)` 로, `isinstance(x, (int, float, int))` 를 `isinstance(x, (int, float))` 로 변경합니다.

itertools_imports
`itertools.ifilter()`, `itertools.izip()`, `itertools.imap()` 임포트를 제거합니다. `itertools.ifilterfalse()` 임포트를 *itertools.filterfalse()* 로 바꿉니다.

itertools
`itertools.ifilter()`, `itertools.izip()`, `itertools.imap()` 를 각각 그것에 맞는 내장 함수로 변경합니다. `itertools.ifilterfalse()` 를 *itertools.filterfalse()* 로 변경합니다.

long
`long` 을 *int* 로 바꿉니다.

map
map() 을 *list* 로 감싸도록 바꿉니다. `map(None, x)` 를 `list(x)` 로 바꿉니다. `from future_builtins import map` 를 사용하면 이 변환자가 비활성화됩니다.

metaclass
구식의 메타 클래스 문법(클래스 바디에 `__metaclass__ = Meta` 를 사용)을 새로운 문법(`class X(metaclass=Meta)`)으로 변경합니다.

methodattrs
구식의 메서드 어트리뷰트 이름을 수정합니다. 예를 들어 `meth.im_func` 를 `meth.__func__` 로 변경합니다.

ne
구식의 부등호 문법인 `<>` 을 `!=` 로 변경합니다.

next
이터레이터의 `next()` 메서드 사용을 *next()* 함수로 변경합니다. 또한 *next()* 메서드를 *__next__()* 로 바꿉니다.

nonzero
__nonzero__() 를 __bool__() 로 변경합니다.

numliterals
8진수 리터럴을 새 문법으로 변경합니다.

operator
operator 모듈에 있는 다양한 함수 호출을 그것에 대응하는 다른 함수 호출로 변경합니다. import collections.abc 와 같이 필요하다면 import 구문도 추가됩니다. 다음과 같이 변경합니다.

변경 전	변경 후
operator.isCallable(obj)	callable(obj)
operator.sequenceIncludes(obj)	operator.contains(obj)
operator.isSequenceType(obj)	isinstance(obj, collections.abc.Sequence)
operator.isMappingType(obj)	isinstance(obj, collections.abc.Mapping)
operator.isNumberType(obj)	isinstance(obj, numbers.Number)
operator.repeat(obj, n)	operator.mul(obj, n)
operator.irepeat(obj, n)	operator.imul(obj, n)

paren
리스트 컴프리헨션 안에 괄호가 필요한 경우 추가합니다. 예를 들어 [x for x in 1, 2] 를 [x for x in (1, 2)] 로 변경합니다.

print
print 구문을 *print()* 함수로 변경합니다.

raise
raise E, V 를 raise E(V) 로, raise E, V, T 를 raise E(V).with_traceback(T) 로 변경합니다. 만약 E 가 튜플인 경우, 변환된 결과물은 동작하지 않을 것입니다. 왜냐하면 튜플이 예외를 대체하는 것은 3.0부터 사라졌기 때문입니다.

raw_input
raw_input() 를 *input()* 로 변경합니다.

reduce
reduce() 를 *functools.reduce()* 로 변경합니다.

reload
reload() 를 *importlib.reload()* 로 변경합니다.

renames
sys.maxint 를 *sys.maxsize* 로 변경합니다.

repr
백틱 repr을 *repr()* 함수로 바꿉니다.

set_literal
set 생성자를 집합 리터럴로 바꿉니다. 이 변환자는 선택적입니다.

standarderror
StandardError 를 *Exception* 로 바꿉니다.

sys_exc
더 사용되지 않을 sys.exc_value, sys.exc_type, sys.exc_traceback 를 *sys.exc_info()* 로 변경합니다.

throw
제너레이터 throw() 메서드의 API 변경 사항을 반영합니다.

tuple_params
묵시적으로 튜플 매개 변수를 언패킹하는 것을 제거합니다. 이로 인해 이 변환자는 임시 변수를 추가합니다.

types
types 모듈에서 몇몇 멤버가 삭제되어 코드가 동작하지 않던 것을 수정합니다.

unicode
unicode 를 *str* 로 변경합니다.

urllib
urllib 와 urllib2 를 *urllib* 패키지로 변경합니다.

ws_comma
쉼표로 구분된 아이템 목록에서 필요 이상의 공백을 제거합니다. 이 변경자는 선택적입니다.

xrange
xrange() 를 *range()* 로 바꿉니다. 기존 *range()* 를 *list* 로 감쌉니다.

xreadlines
for x in file.xreadlines() 를 for x in file 로 변경합니다.

zip
zip() 를 *list* 로 감쌉니다. 이 변환자는 from future_builtins import zip 가 있을 때는 비활성화됩니다.

27.7.3 lib2to3 - 2to3 라이브러리

소스 코드: Lib/lib2to3/

참고: *lib2to3* API는 미래에 크게 바뀔 수 있기 때문에 안정적이지 않다고 생각해야 합니다.

27.8 test — Regression tests package for Python

참고: The *test* package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The *test* package contains all regression tests for Python as well as the modules *test.support* and test.regrtest. *test.support* is used to enhance your tests while test.regrtest drives the testing suite.

Each module in the *test* package whose name starts with test_ is a testing suite for a specific module or feature. All new tests should be written using the *unittest* or *doctest* module. Some older tests are written using a "traditional" testing style that compares output printed to sys.stdout; this style of test is considered deprecated.

더 보기:

Module *unittest* Writing PyUnit regression tests.

Module *doctest* Tests embedded in documentation strings.

27.8.1 Writing Unit Tests for the `test` package

It is preferred that tests that use the *unittest* module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by `test.regrtest`, on its own as a script that supports the *unittest* CLI, or via the `python -m unittest` CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also "private" code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.

- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```
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)
```

  When using this pattern, remember that all classes that inherit from *unittest.TestCase* are run as tests. The `Mixin` class in the example above does not have any data and so can't be run by itself, thus it does not inherit from *unittest.TestCase*.

더 보기:

Test Driven Development A book by Kent Beck on writing tests before code.

27.8.2 Running tests using the command-line interface

The *test* package can be run as a script to drive Python's regression test suite, thanks to the `-m` option: **python -m test**. Under the hood, it uses `test.regrtest`; the call **python -m test.regrtest** used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the *test* package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present or loading the tests via unittest.TestLoader.loadTestsFromModule if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (**python -m test test_spam**) will minimize output and only print whether the test passed or failed.

Running *test* directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: **python -m test -uall**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **python -m test -uall,-audio,-largefile** will run *test* with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **python -m test -h**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **make test** at the top-level directory where Python was built. On Windows, executing **rt.bat** from your `PCbuild` directory will run all regression tests.

27.9 test.support — Utilities for the Python test suite

The *test.support* module provides support for Python's regression test suite.

참고: *test.support* is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions:

exception test.support.**TestFailed**
Exception to be raised when a test fails. This is deprecated in favor of *unittest*-based tests and *unittest.TestCase*'s assertion methods.

exception test.support.**ResourceDenied**
Subclass of *unittest.SkipTest*. Raised when a resource (such as a network connection) is not available. Raised by the *requires()* function.

The *test.support* module defines the following constants:

test.support.**verbose**
`True` when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by `test.regrtest`.

test.support.**is_jython**
`True` if the running interpreter is Jython.

test.support.**is_android**
`True` if the system is Android.

test.support.**unix_shell**
Path for shell if not on Windows; otherwise `None`.

test.support.**FS_NONASCII**
A non-ASCII character encodable by *os.fsencode()*.

test.support.**TESTFN**
Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.**TESTFN_UNICODE**
Set to a non-ASCII name for a temporary file.

test.support.**TESTFN_ENCODING**
Set to *sys.getfilesystemencoding()*.

test.support.**TESTFN_UNENCODABLE**
Set to a filename (str type) that should not be able to be encoded by file system encoding in strict mode. It may be `None` if it's not possible to generate such a filename.

test.support.**TESTFN_UNDECODABLE**
Set to a filename (bytes type) that should not be able to be decoded by file system encoding in strict mode. It may be `None` if it's not possible to generate such a filename.

test.support.**TESTFN_NONASCII**
Set to a filename containing the *FS_NONASCII* character.

test.support.**IPV6_ENABLED**
Set to `True` if IPV6 is enabled on this host, `False` otherwise.

test.support.**SAVEDCWD**
Set to *os.getcwd()*.

test.support.**PGO**
Set when tests can be skipped when they are not useful for PGO.

test.support.**PIPE_MAX_SIZE**
A constant that is likely larger than the underlying OS pipe buffer size, to make writes blocking.

test.support.**SOCK_MAX_SIZE**
A constant that is likely larger than the underlying OS socket buffer size, to make writes blocking.

test.support.**TEST_SUPPORT_DIR**
Set to the top level directory that contains *test.support*.

test.support.**TEST_HOME_DIR**
Set to the top level directory for the test package.

test.support.**TEST_DATA_DIR**
Set to the `data` directory within the test package.

test.support.**MAX_Py_ssize_t**
Set to *sys.maxsize* for big memory tests.

test.support.**max_memuse**
Set by *set_memlimit()* as the memory limit for big memory tests. Limited by *MAX_Py_ssize_t*.

test.support.**real_max_memuse**
Set by *set_memlimit()* as the memory limit for big memory tests. Not limited by *MAX_Py_ssize_t*.

test.support.**MISSING_C_DOCSTRINGS**
Return `True` if running on CPython, not on Windows, and configuration not set with `WITH_DOC_STRINGS`.

test.support.**HAVE_DOCSTRINGS**
Check for presence of docstrings.

test.support.**TEST_HTTP_URL**
Define the URL of a dedicated HTTP server for the network tests.

test.support.**ALWAYS_EQ**
Object that is equal to anything. Used to test mixed type comparison.

test.support.**LARGEST**
Object that is greater than anything (except itself). Used to test mixed type comparison.

test.support.**SMALLEST**
Object that is less than anything (except itself). Used to test mixed type comparison.

The *test.support* module defines the following functions:

test.support.**forget**(*module_name*)
Remove the module named *module_name* from `sys.modules` and delete any byte-compiled files of the module.

test.support.**unload**(*name*)
Delete *name* from `sys.modules`.

test.support.**unlink**(*filename*)
Call *os.unlink()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence fo the file.

test.support.**rmdir**(*filename*)
Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

test.support.**rmtree**(*path*)
Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the files.

test.support.**make_legacy_pyc**(*source*)
Move a PEP 3147/488 pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The *source* value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

test.support.**is_resource_enabled**(*resource*)
Return `True` if *resource* is enabled and available. The list of available resources is only set when `test.regrtest` is executing the tests.

test.support.**python_is_optimized**()
Return `True` if Python was not built with `-O0` or `-Og`.

test.support.**with_pymalloc**()
Return `_testcapi.WITH_PYMALLOC`.

test.support.**requires**(*resource*, *msg=None*)
Raise *ResourceDenied* if *resource* is not available. *msg* is the argument to *ResourceDenied* if it is raised. Always returns `True` if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by `test.regrtest`.

test.support.**system_must_validate_cert**(*f*)
Raise *unittest.SkipTest* on TLS certification validation failures.

test.support.**sortdict**(*dict*)
Return a repr of *dict* with keys sorted.

test.support.**findfile**(*filename*, *subdir=None*)
Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

Setting *subdir* indicates a relative path to use to find the file rather than looking directly in the path directories.

test.support.**create_empty_file**(*filename*)
Create an empty file with *filename*. If it already exists, truncate it.

test.support.**fd_count**()
Count the number of open file descriptors.

test.support.**match_test**(*test*)
Match *test* to patterns set in *set_match_tests()*.

test.support.**set_match_tests**(*patterns*)
Define match test with regular expression *patterns*.

test.support.**run_unittest**(**classes*)
Execute *unittest.TestCase* subclasses passed to the function. The function scans the classes for methods starting with the prefix `test_` and executes the tests individually.

It is also legal to pass strings as parameters; these should be keys in `sys.modules`. Each associated module will be scanned by `unittest.TestLoader.loadTestsFromModule()`. This is usually seen in the following `test_main()` function:

```
def test_main():
    support.run_unittest(__name__)
```

This will run all tests defined in the named module.

test.support.**run_doctest**(*module*, *verbosity=None*, *optionflags=0*)
Run *doctest.testmod()* on the given *module*. Return `(failure_count, test_count)`.

If *verbosity* is `None`, *doctest.testmod()* is run with verbosity set to *verbose*. Otherwise, it is run with verbosity set to `None`. *optionflags* is passed as `optionflags` to *doctest.testmod()*.

test.support.**setswitchinterval**(*interval*)
Set the *sys.setswitchinterval()* to the given *interval*. Defines a minimum interval for Android systems to prevent the system from hanging.

test.support.**check_impl_detail**(***guards*)
Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments:

```
check_impl_detail()               # Only on CPython (default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except CPython.
```

test.support.**check_warnings**(**filters*, *quiet=True*)
A convenience wrapper for *warnings.catch_warnings()* that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with *warnings.simplefilter()* set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is `False`, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to `True`.

If no arguments are specified, it defaults to:

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from *catch_warnings()* is available via the recorder object's *warnings* attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return `None`.

The recorder object also has a `reset()` method, which clears the warnings list.

The context manager is designed to be used like this:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, *check_warnings()* would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

버전 3.2에서 변경: New optional arguments *filters* and *quiet*.

test.support.**check_no_resource_warning**(*testcase*)
Context manager to check that no *ResourceWarning* was raised. You must remove the object which may emit *ResourceWarning* before the end of the context manager.

test.support.**set_memlimit**(*limit*)
Set the values for *max_memuse* and *real_max_memuse* for big memory tests.

test.support.**record_original_stdout**(*stdout*)
Store the value from *stdout*. It is meant to hold the stdout at the time the regrtest began.

test.support.**get_original_stdout**()
Return the original stdout set by *record_original_stdout()* or `sys.stdout` if it's not set.

test.support.**strip_python_strerr**(*stderr*)
Strip the *stderr* of a Python process from potential debug output emitted by the interpreter. This will typically be run on the result of *subprocess.Popen.communicate()*.

test.support.**args_from_interpreter_flags**()
Return a list of command line arguments reproducing the current settings in `sys.flags` and `sys.warnoptions`.

test.support.**optim_args_from_interpreter_flags**()
Return a list of command line arguments reproducing the current optimization settings in `sys.flags`.

test.support.**captured_stdin**()
test.support.**captured_stdout**()
test.support.**captured_stderr**()
A context managers that temporarily replaces the named stream with *io.StringIO* object.

Example use with output streams:

```
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"
```

Example use with input stream:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**temp_dir**(*path=None*, *quiet=False*)
A context manager that creates a temporary directory at *path* and yields the directory.

If *path* is `None`, the temporary directory is created using *tempfile.mkdtemp()*. If *quiet* is `False`, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

`test.support.`**`change_cwd`**(*path*, *quiet=False*)
A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is `False`, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

`test.support.`**`temp_cwd`**(*name='tempcwd'*, *quiet=False*)
A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is `None`, the temporary directory is created using *tempfile.mkdtemp()*.

If *quiet* is `False` and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

`test.support.`**`temp_umask`**(*umask*)
A context manager that temporarily sets the process umask.

`test.support.`**`transient_internet`**(*resource_name*, ***, *timeout=30.0*, *errnos=()*)
A context manager that raises *ResourceDenied* when various issues with the internet connection manifest themselves as exceptions.

`test.support.`**`disable_faulthandler`**()
A context manager that replaces `sys.stderr` with `sys.__stderr__`.

`test.support.`**`gc_collect`**()
Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that `__del__` methods may be called later than expected and weakrefs may remain alive for longer than expected.

`test.support.`**`disable_gc`**()
A context manager that disables the garbage collector upon entry and reenables it upon exit.

`test.support.`**`swap_attr`**(*obj*, *attr*, *new_val*)
Context manager to swap out an attribute with a new object.

Usage:

```
with swap_attr(obj, "attr", 5):
    ...
```

This will set `obj.attr` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `attr` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

`test.support.`**`swap_item`**(*obj*, *attr*, *new_val*)
Context manager to swap out an item with a new object.

Usage:

```
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

test.support.**wait_threads_exit**(*timeout=60.0*)
Context manager to wait until all threads created in the `with` statement exit.

test.support.**start_threads**(*threads*, *unlock=None*)
Context manager to start *threads*. It attempts to join the threads upon exit.

test.support.**calcobjsize**(*fmt*)
Return *struct.calcsize()* for `nP{fmt}0n` or, if `gettotalrefcount` exists, `2PnP{fmt}0P`.

test.support.**calcvobjsize**(*fmt*)
Return *struct.calcsize()* for `nPn{fmt}0n` or, if `gettotalrefcount` exists, `2PnPn{fmt}0P`.

test.support.**checksizeof**(*test*, *o*, *size*)
For testcase *test*, assert that the `sys.getsizeof` for *o* plus the GC header size equals *size*.

test.support.**can_symlink**()
Return `True` if the OS supports symbolic links, `False` otherwise.

test.support.**can_xattr**()
Return `True` if the OS supports xattr, `False` otherwise.

@test.support.**skip_unless_symlink**
A decorator for running tests that require support for symbolic links.

@test.support.**skip_unless_xattr**
A decorator for running tests that require support for xattr.

@test.support.**skip_unless_bind_unix_socket**
A decorator for running tests that require a functional bind() for Unix sockets.

@test.support.**anticipate_failure**(*condition*)
A decorator to conditionally mark tests with *unittest.expectedFailure()*. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

@test.support.**run_with_locale**(*catstr*, **locales*)
A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example `"LC_ALL"`). The *locales* passed will be tried sequentially, and the first valid locale will be used.

@test.support.**run_with_tz**(*tz*)
A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

@test.support.**requires_freebsd_version**(**min_version*)
Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_linux_version**(**min_version*)
Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_mac_version**(**min_version*)
Decorator for the minimum version when running test on Mac OS X. If the MAC OS X version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_IEEE_754**
Decorator for skipping tests on non-IEEE 754 platforms.

@test.support.**requires_zlib**
Decorator for skipping tests if *zlib* doesn't exist.

@test.support.**requires_gzip**
Decorator for skipping tests if *gzip* doesn't exist.

@test.support.**requires_bz2**
Decorator for skipping tests if *bz2* doesn't exist.

@test.support.**requires_lzma**
Decorator for skipping tests if *lzma* doesn't exist.

@test.support.**requires_resource**(*resource*)
Decorator for skipping tests if *resource* is not available.

@test.support.**requires_docstrings**
Decorator for only running the test if *HAVE_DOCSTRINGS*.

@test.support.**cpython_only**(*test*)
Decorator for tests only applicable to CPython.

@test.support.**impl_detail**(*msg=None*, ***guards*)
Decorator for invoking *check_impl_detail()* on *guards*. If that returns `False`, then uses *msg* as the reason for skipping the test.

@test.support.**no_tracing**(*func*)
Decorator to temporarily turn off tracing for the duration of the test.

@test.support.**refcount_test**(*test*)
Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.

@test.support.**reap_threads**(*func*)
Decorator to ensure the threads are cleaned up even if the test fails.

@test.support.**bigmemtest**(*size*, *memuse*, *dry_run=True*)
Decorator for bigmem tests.

size is a requested size for the test (in arbitrary, test-interpreted units.) *memuse* is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with `@bigmemtest(size=_4G, memuse=2)`.

The *size* argument is normally passed to the decorated test method as an extra argument. If *dry_run* is `True`, the value passed to the test method may be less than the requested value. If *dry_run* is `False`, it means the test doesn't support dummy runs when `-M` is not specified.

@test.support.**bigaddrspacetest**(*f*)
Decorator for tests that fill the address space. *f* is the function to wrap.

test.support.**make_bad_fd**()
Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.**check_syntax_error**(*testcase*, *statement*, *errtext=''*, ***, *lineno=None*, *offset=None*)
Test for syntax errors in *statement* by attempting to compile *statement*. *testcase* is the *unittest* instance for the test. *errtext* is the text of the error raised by *SyntaxError*. If *lineno* is not None, compares to the line of the *SyntaxError*. If *offset* is not None, compares to the offset of the *SyntaxError*.

test.support.**open_urlresource**(*url*, **args*, ***kw*)
Open *url*. If open fails, raises *TestFailed*.

test.support.**import_module**(*name*, *deprecated=False*, ***, *required_on()*)
This function imports and returns the named module. Unlike a normal import, this function raises *unittest.SkipTest* if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`. If a module is required on a platform but optional for others, set *required_on* to an iterable of platform prefixes which will be compared against *sys.platform*.

버전 3.1에 추가.

test.support.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)
This function imports and returns a fresh copy of the named Python module by removing the named module from sys.modules before doing the import. Note that unlike reload(), the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the sys.modules cache before doing the import.

blocked is an iterable of module names that are replaced with None in the module cache during the import to ensure that attempts to import them raise *ImportError*.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into sys.modules when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is True.

This function will raise *ImportError* if the named module cannot be imported.

Example use:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

버전 3.1에 추가.

test.support.**modules_setup**()
Return a copy of *sys.modules*.

test.support.**modules_cleanup**(*oldmodules*)
Remove modules except for *oldmodules* and encodings in order to preserve internal cache.

test.support.**threading_setup**()
Return current thread count and copy of dangling threads.

test.support.**threading_cleanup**(**original_values*)
Cleanup up threads not specified in *original_values*. Designed to emit a warning if a test leaves running threads in the background.

test.support.**join_thread**(*thread*, *timeout=30.0*)
Join a *thread* within *timeout*. Raise an *AssertionError* if thread is still alive after *timeout* seconds.

test.support.**reap_children**()
Use this at the end of test_main whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.**get_attribute**(*obj*, *name*)
Get an attribute, raising *unittest.SkipTest* if *AttributeError* is raised.

test.support.**bind_port**(*sock*, *host=HOST*)
Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the sock.family is *AF_INET* and sock.type is *SOCK_STREAM*, and the socket has SO_REUSEADDR or SO_REUSEPORT set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the SO_EXCLUSIVEADDRUSE socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

test.support.**bind_unix_socket**(*sock*, *addr*)
Bind a unix socket, raising *unittest.SkipTest* if *PermissionError* is raised.

test.support.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)
Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is *AF_INET*, *SOCK_STREAM*), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or *bind_port()* should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's s_server mode). Always prefer *bind_port()* over *find_unused_port()* where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)
Generic implementation of the *unittest* `load_tests` protocol for use in test packages. *pkg_dir* is the root directory of the package; *loader*, *standard_tests*, and *pattern* are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**fs_is_case_insensitive**(*directory*)
Return `True` if the file system for *directory* is case-insensitive.

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)
Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with '_' but includes all magic methods, i.e. those starting and ending in '__'.

버전 3.5에 추가.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)
Override *object_to_patch.attr_name* with *new_value*. Also add cleanup procedure to *test_instance* to restore *object_to_patch* for *attr_name*. The *attr_name* should be a valid attribute for *object_to_patch*.

test.support.**run_in_subinterp**(*code*)
Run *code* in subinterpreter. Raise *unittest.SkipTest* if *tracemalloc* is enabled.

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)
Assert that *iter* is deallocated after iterating.

test.support.**missing_compiler_executable**(*cmd_names=[]*)
Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or `None` when none is found missing.

test.support.**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *blacklist=()*)
Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other

modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones.

The *blacklist* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

Example use:

```
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'}
        blacklist = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, blacklist=blacklist)
```

버전 3.6에 추가.

`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.

버전 3.7.14에 추가.

The *test.support* module defines the following classes:

class `test.support.`**`TransientResource`**(*exc*, ***kwargs*)
Instances are a context manager that raises *ResourceDenied* if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the `with` statement. Only if all pairs match properly against attributes on the exception is *ResourceDenied* raised.

class `test.support.`**`EnvironmentVarGuard`**
Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying `os.environ`. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

버전 3.1에서 변경: Added dictionary interface.

`EnvironmentVarGuard.`**`set`**(*envvar*, *value*)
Temporarily set the environment variable `envvar` to the value of `value`.

`EnvironmentVarGuard.`**`unset`**(*envvar*)
Temporarily unset the environment variable `envvar`.

class `test.support.`**`SuppressCrashReport`**
A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetErrorMode.

On UNIX, *resource.setrlimit()* is used to set *resource.RLIMIT_CORE*'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class `test.support.`**CleanImport**(**module_names*)
A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a DeprecationWarning on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class `test.support.`**DirsOnSysPath**(**paths*)
A context manager to temporarily add directories to sys.path.

This makes a copy of *sys.path*, appends any directories given as positional arguments, then reverts *sys.path* to the copied settings when the context ends.

Note that *all* *sys.path* modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

class `test.support.`**SaveSignals**
Class to save and restore signal handlers registered by the Python signal handler.

class `test.support.`**Matcher**

matches(*self*, *d*, ***kwargs*)
Try to match a single dict with the supplied arguments.

match_value(*self*, *k*, *dv*, *v*)
Try to match a single stored value (*dv*) with a supplied value (*v*).

class `test.support.`**WarningsRecorder**
Class used to record warnings for unit tests. See documentation of *check_warnings()* above for more details.

class `test.support.`**BasicTestRunner**

run(*test*)
Run *test* and return the result.

class `test.support.`**TestHandler**(*logging.handlers.BufferingHandler*)
Class for logging support.

class `test.support.`**FakePath**(*path*)
Simple *path-like object*. It implements the `__fspath__()` method which just returns the *path* argument. If *path* is an exception, it will be raised in `__fspath__()`.

27.10 `test.support.script_helper` — Utilities for the Python execution tests

The *test.support.script_helper* module provides support for Python's script execution tests.

`test.support.script_helper.`**interpreter_requires_environment**()
Return `True` if `sys.executable interpreter` requires environment variables in order to be able to run at all.

This is designed to be used with `@unittest.skipIf()` to annotate tests that need to use an `assert_python*()` function to launch an isolated mode (`-I`) or no environment mode (`-E`) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn't have an obvious home with Python's current home finding logic.

Setting `PYTHONHOME` is one way to get most of the testsuite to run in that situation. `PYTHONPATH` or `PYTHONUSERSITE` are other common environment variables that might impact whether or not the interpreter can start.

`test.support.script_helper.`**`run_python_until_end`**(**args*, ***env_vars*)
Set up the environment based on *env_vars* for running the interpreter in a subprocess. The values can include `__isolated`, `__cleanenv`, `__cwd`, and `TERM`.

`test.support.script_helper.`**`assert_python_ok`**(**args*, ***env_vars*)
Assert that running the interpreter with *args* and optional environment variables *env_vars* succeeds (`rc == 0`) and return a `(return code, stdout, stderr)` tuple.

If the `__cleanenv` keyword is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the `__isolated` keyword is set to `False`.

`test.support.script_helper.`**`assert_python_failure`**(**args*, ***env_vars*)
Assert that running the interpreter with *args* and optional environment variables *env_vars* fails (`rc != 0`) and return a `(return code, stdout, stderr)` tuple.

See *assert_python_ok()* for more options.

`test.support.script_helper.`**`spawn_python`**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)
Run a Python subprocess with the given arguments.

kw is extra keyword args to pass to *subprocess.Popen()*. Returns a *subprocess.Popen* object.

`test.support.script_helper.`**`kill_python`**(*p*)
Run the given *subprocess.Popen* process until completion and return stdout.

`test.support.script_helper.`**`make_script`**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)
Create script containing *source* in path *script_dir* and *script_basename*. If *omit_suffix* is `False`, append `.py` to the name. Return the full script path.

`test.support.script_helper.`**`make_zip_script`**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)
Create zip file at *zip_dir* and *zip_basename* with extension `zip` which contains the files in *script_name*. *name_in_zip* is the archive name. Return a tuple containing `(full path, full path of archive name)`.

`test.support.script_helper.`**`make_pkg`**(*pkg_dir*, *init_source=''*)
Create a directory named *pkg_dir* containing an `__init__` file with *init_source* as its contents.

`test.support.script_helper.`**`make_zip_pkg`**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)
Create a zip package directory with a path of *zip_dir* and *zip_basename* containing an empty `__init__` file and a file *script_basename* containing the *source*. If *compiled* is `True`, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.

파이썬 개발 모드도 보십시오: `-X dev` 옵션과 `PYTHONDEVMODE` 환경 변수.

CHAPTER 28

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.

28.1 bdb — Debugger framework

Source code: Lib/bdb.py

The *bdb* module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

exception bdb.**BdbQuit**
Exception raised by the *Bdb* class for quitting the debugger.

The *bdb* module also defines two classes:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=0*, *cond=None*, *funcname=None*)
This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called `bpbynumber` and by `(file, line)` pairs through `bplist`. The former points to a single instance of class *Breakpoint*. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a *funcname* is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

Breakpoint instances have the following methods:

deleteMe()
Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()
Mark the breakpoint as enabled.

disable()
Mark the breakpoint as disabled.

bpformat()
Return a string with all the information about the breakpoint, nicely formatted:

- The breakpoint number.
- If it is temporary or not.
- Its file,line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

버전 3.2에 추가.

bpprint(*out=None*)
Print the output of *bpformat()* to the file *out*, or if it is None, to standard output.

class bdb.**Bdb**(*skip=None*)
The *Bdb* class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (*pdb.Pdb*) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the __name__ in the frame globals.

버전 3.1에 추가: The *skip* argument.

The following methods of *Bdb* normally don't need to be overridden.

canonic(*filename*)
Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()
Set the botframe, stopframe, returnframe and quitting attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)
This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- "line": A new line of code is going to be executed.
- "call": A function is about to be called, or another code block entered.
- "return": A function or other code block is about to return.
- "exception": An exception has occurred.
- "c_call": A C function is about to be called.
- "c_return": A C function has returned.

- `"c_exception"`: A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for *sys.settrace()* for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(*frame*)
If the debugger should stop on the current line, invoke the *user_line()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_line()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_call(*frame*, *arg*)
If the debugger should stop on this function call, invoke the *user_call()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_call()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_return(*frame*, *arg*)
If the debugger should stop on this function return, invoke the *user_return()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_return()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)
If the debugger should stop at this exception, invokes the *user_exception()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_exception()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

stop_here(*frame*)
This method checks if the *frame* is somewhere below `botframe` in the call stack. `botframe` is the frame in which debugging started.

break_here(*frame*)
This method checks if there is a breakpoint in the filename and line belonging to *frame* or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

break_anywhere(*frame*)
This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)
This method is called from *dispatch_call()* when there is the possibility that a break might be necessary anywhere inside the called function.

user_line(*frame*)
This method is called from *dispatch_line()* when either *stop_here()* or *break_here()* yields `True`.

user_return(*frame*, *return_value*)
This method is called from *dispatch_return()* when *stop_here()* yields `True`.

user_exception(*frame*, *exc_info*)
This method is called from *dispatch_exception()* when *stop_here()* yields `True`.

do_clear(*arg*)
Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()
Stop after one line of code.

set_next(*frame*)
Stop on the next line in or below the given frame.

set_return(*frame*)
Stop when returning from the given frame.

set_until(*frame*)
Stop when the line with the line no greater than the current one is reached or when returning from current frame.

set_trace([*frame*])
Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

set_continue()
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()
Set the `quitting` attribute to `True`. This raises *BdbQuit* in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=0*, *cond*, *funcname*)
Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the *canonic()* method.

clear_break(*filename*, *lineno*)
Delete the breakpoints in *filename* and *lineno*. If none were set, an error message is returned.

clear_bpbynumber(*arg*)
Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)
Delete all breakpoints in *filename*. If none were set, an error message is returned.

clear_all_breaks()
Delete all existing breakpoints.

get_bpbynumber(*arg*)
Return a breakpoint specified by the given number. If *arg* is a string, it will be converted to a number. If *arg* is a non-numeric string, if the given breakpoint never existed or has been deleted, a *ValueError* is raised.

버전 3.2에 추가.

get_break(*filename*, *lineno*)
Check if there is a breakpoint for *lineno* of *filename*.

get_breaks(*filename*, *lineno*)
Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)
Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(*frame_lineno*, *lprefix=': '*)
Return a string with information about a stack entry, identified by a `(frame, lineno)` tuple:

- The canonical form of the filename which contains the frame.
- The function name, or `"<lambda>"`.
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

run(*cmd*, *globals=None*, *locals=None*)
Debug a statement executed via the *exec()* function. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

runeval(*expr*, *globals=None*, *locals=None*)
Debug an expression executed via the *eval()* function. *globals* and *locals* have the same meaning as in *run()*.

runctx(*cmd*, *globals*, *locals*)
For backwards compatibility. Calls the *run()* method.

runcall(*func*, **args*, ***kwds*)
Debug a single function call, and return its result.

Finally, the module defines the following functions:

bdb.**checkfuncname**(*b*, *frame*)
Check whether we should break here, depending on the way the breakpoint *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

bdb.**effective**(*file*, *line*, *frame*)
Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return `(None, None)` if there is no matching breakpoint.

bdb.**set_trace**()
Start debugging with a *Bdb* instance from caller's frame.

28.2 faulthandler — 파이썬 트레이스백 덤프

버전 3.3에 추가.

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.

28.2.1 트레이스백 덤프하기

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)
모든 스레드의 트레이스백을 *file*로 덤프합니다. *all_threads*가 False면, 현재 스레드만 덤프합니다.

버전 3.5에서 변경: 이 함수에 파일 기술자를 전달하는 지원이 추가되었습니다.

28.2.2 결함 처리기 상태

faulthandler.**enable**(*file=sys.stderr*, *all_threads=True*)
결함 처리기를 활성화합니다: SIGSEGV, SIGFPE, SIGABRT, SIGBUS 및 SIGILL 시그널에 대한 처리기를 설치하여 파이썬 트레이스백을 덤프합니다. *all_threads*가 True면 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.

버전 3.5에서 변경: 이 함수에 파일 기술자를 전달하는 지원이 추가되었습니다.

버전 3.6에서 변경: 윈도우에서는, 윈도우 예외(Windows exception) 처리기도 설치됩니다.

faulthandler.**disable**()
결함 처리기를 비활성화합니다: *enable()*로 설치된 시그널 처리기를 제거합니다.

faulthandler.**is_enabled**()
결함 처리기가 활성화되었는지 검사합니다.

28.2.3 시간 초과 후에 트레이스백 덤프하기

faulthandler.**dump_traceback_later**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)
timeout 초의 시간제한 후, 또는 *repeat*가 True면 매 *timeout* 초마다, 모든 스레드의 트레이스백을 덤프합니다. *exit*가 True면, 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.

This function is implemented using a watchdog thread.

버전 3.7에서 변경: This function is now always available.

버전 3.5에서 변경: 이 함수에 파일 기술자를 전달하는 지원이 추가되었습니다.

faulthandler.**cancel_dump_traceback_later**()
마지막 *dump_traceback_later()* 호출을 취소합니다.

28.2.4 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 False면 현재 스레드의, 트레이스백을 *file*로 덤프합니다. chain이 True면 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 True를 반환하고, 그렇지 않으면 False를 반환합니다.

윈도우에서는 사용할 수 없습니다.

28.2.5 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.

28.2.6 예제

리눅스에서 결함 처리기를 활성화하거나 그렇지 않았을 때의 세그멘테이션 결함 예제:

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

28.3 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.

디버거의 프롬프트는 `(Pdb)` 입니다. 디버거 제어하에 프로그램을 실행하는 일반적인 사용법은 다음과 같습니다:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

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.

`pdb.py` 는 다른 스크립트를 디버그하기 위한 스크립트로 호출될 수 있습니다. 예를 들면:

```
python3 -m pdb myscript.py
```

스크립트로 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.

버전 3.2에 추가: `pdb.py` 는 이제 `.pdbrc` 파일에 주어진 것처럼 명령을 실행하는 `-c` 옵션을 받을 수 있습니다, 디버거 명령들 을 확인해보세요.

버전 3.7에 추가: `pdb.py` 는 이제 `python3 -m`과 비슷한 모듈을 실행하는 `-m` 옵션을 받을 수 있습니다. 스크립트와 마찬가지로, 디버거는 모듈의 첫 번째 줄 바로 전에 실행을 일시정지합니다.

실행 중인 프로그램에서 디버거로 진입하는 일반적인 사용법은:

```
import pdb; pdb.set_trace()
```

위 코드를 디버거로 진입하고 싶은 구간에 추가하면 됩니다. 그런 다음 이 문장 뒤에 오는 코드를 단계별로 실행하고, *continue* 명령을 사용하여 디버거 없이 프로그램을 계속 실행할 수 있습니다.

버전 3.7에 추가: 내장 *breakpoint()* 가, 기본값으로 호출될 때는, `import pdb; pdb.set_trace()` 를 대신해서 사용할 수 있습니다.

에러가 발생하는 프로그램을 검사하는 일반적인 사용법은 다음과 같습니다:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

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, 표현식의 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*)
: 호출하는 스택 프레임에서 디버거에 진입합니다. 코드가 디버그 되지 않는 경우 일지라도 (예를 들면, assertion이 실패하는 경우), 프로그램의 특정 지점에 중단점을 하드 코딩할 때 유용하게 사용됩니다. *header* 값을 주면, 디버깅이 시작되기 바로 전에 그 값이 콘솔에 출력됩니다.

  버전 3.7에서 변경: 키워드 전용 인자 *header*.

pdb.**post_mortem**(*traceback=None*)
: 주어진 *traceback* 객체의 포스트-모템(post-mortem) 디버깅으로 진입합니다. 만약 *traceback*이 주어지지 않았다면, 현재 처리되고 있는 하나의 예외를 사용합니다. (기본값을 사용하는 경우 예외는 반드시 처리되고 있어야 합니다.)

pdb.**pm**()
: *sys.last_traceback* 에서 찾은 traceback 의 포스트-모템(post-mortem) 디버깅으로 진입합니다.

`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*)

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.

*skip*으로 추적하기 위한 호출 예시:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

버전 3.1에 추가: *skip* 인자.

버전 3.2에 추가: *nosigint* 인자. 이전에는, Pdb가 SIGINT 핸들러를 설정하지 않았습니다.

버전 3.6에서 변경: *readrc* 인자.

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.

28.3.1 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, 예외명은 출력되지만 디버거의 상태는 변경되지 않습니다.

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.

만약 파일 `.pdbrc` 가 사용자의 홈 디렉터리 또는 현재 디렉터리에 있으면, 디버거 프롬프트에서 입력된 것처럼 읽히고 실행됩니다. 이것은 특히 에일리어스에 유용합니다. 만약 두 파일이 모두 존재하면, 홈 디렉터리에 있는 파일이 먼저 읽히고 거기에 정의된 에일리어스는 로컬 파일에 의해 무시될 수 있습니다.

[1] 프레임이 특정 모듈에서 시작되는 것으로 간주하는지 여부는 프레임 전역에 있는 `__name__`에 의해 결정됩니다.

버전 3.2에서 변경: .pdbrc 는 *continue*와 *next*같이 디버깅을 계속하는 명령을 포함할 수 있습니다. 이전에는, 이런 명령이 효과가 없었습니다.

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)
가장 최근 프레임을 맨 아래로 하는 스택 트레이스를 출력합니다. 화살표는 현재 프레임을 나타내며, 대부분의 명령의 컨텍스트를 명확히 합니다.

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]]
lineno 인자가 주어진 경우, 현재 파일의 해당 줄 번호에 브레이크를 설정합니다. *function* 인자가 주어진 경우, 함수 내에서 첫 번째 실행 가능한 문장에서 브레이크를 설정합니다. 줄 번호는 다른 파일 (아마도 아직 로드되지 않은 파일)에 중단점을 지정하기 위해, 파일명과 콜론을 접두사로 사용할 수 있습니다. 파일은 *sys.path* 에서 검색합니다. 주의할 점은 각 중단점에 번호가 지정되며, 다른 모든 중단점 명령이 그 번호를 참조하게 됩니다.

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.

tbreak [([filename:]lineno | function) [, condition]]
한번 도달하면 제거되는 임시중단점입니다. 인자는 *break*과 동일합니다.

cl(ear) [filename:lineno | bpnumber [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 횟수가 생략된 경우, 무시 횟수는 0으로 설정됩니다. 무시 횟수가 0일 때 중단점이 활성화됩니다. 0이 아닐 때는, 중단점에 도달하고 중단점이 비활성화되지 않고 연관 조건이 참일 때마다 그 횟수가 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 end만 입력하면 됩니다. 예를 들면:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

중단점에 지정된 모든 명령을 제거하려면, commands 입력 후에 바로 end를 입력하면 됩니다; 즉, 아무 명령을 설정하지 않는 것입니다.

bpnumber 인자가 주어지지 않으면, commands는 마지막 중단점 묶음을 참조하게 됩니다.

중단점 명령을 활용해서 프로그램을 다시 시작할 수도 있습니다. *continue* 명령이나, *step* 또는 실행을 재개하는 다른 명령을 사용하기만 하면 됩니다.

실행을 재개하는 아무 명령 (*continue*, *step*, *next*, *return*, *jump*, *quit* 및 해당 명령의 약어들)을 지정하는 것은 (end 명령을 붙인 것처럼) 해당 명령 목록을 끝내는 것입니다. 왜냐하면 실행을 재개할 때마다, 명령 목록을 가진 다른 중단점을 맞이할 수 있고, 어떤 목록을 실행해야 할지 모르는 상황이 생기기 때문입니다.

만약 'slient' 명령을 사용하면, 중단점에서 멈출 때 나오는 메시지는 출력되지 않습니다. 특정 메시지를 출력하고 진행되는 중단점에 바람직할 수 있습니다. 다만 그 어떤 명령도 출력하지 않는다면, 그 중단점에 도달했다는 것은 알 수 없습니다.

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.

줄 번호가 주어진 경우에는, 해당 번호보다 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 3.2에 추가: >>표시

ll | longlist
현재 함수나 프레임의 소스 코드 전체를 나열합니다. 참고할만한 줄은 *list*처럼 표시됩니다.

버전 3.2에 추가.

a(rgs)
현재 함수의 인자 목록을 출력합니다.

p expression
현재 컨텍스트에서 *expression*을 실행하고 값을 출력합니다.

참고: 디버거 명령은 아니지만, `print()`도 사용될 수 있습니다 — 이때는 파이썬의 *print()* 함수를 실행하게 됩니다.

---

pp expression
p 명령과 비슷하지만, 표현식의 값을 *pprint* 모듈을 활용하여 보기 좋은 형태로 출력합니다.

whatis expression
*expression*의 형(type)을 출력합니다.

source expression
주어진 객체의 소스 코드를 가져와서 보여줍니다.

버전 3.2에 추가.

display [expression]
현재 프레임에서 실행이 중지될 때마다, 표현식의 값이 변경된 경우 표시합니다.

표현식이 주어지지 않은 경우, 현재 프레임에서 표시되는 모든 표현식을 나열합니다.

버전 3.2에 추가.

undisplay [expression]
현재 프레임에서 표현식을 더는 표시하지 않습니다. 표현식이 주어지지 않은 경우, 현재 프레임에서 표시되는 모든 표현식을 제거합니다.

버전 3.2에 추가.

interact
현재 스코프에서 찾을 수 있는 모든 지역 또는 전역 이름을 담고 있는 전역 이름 공간을 가진 (*code* 모듈을 활용하는) 대화형 인터프리터를 시작합니다.

버전 3.2에 추가.

alias [name [command]]
*command*를 실행하는 *name*이라 불리는 에일리어스를 생성합니다. 명령은 따옴표로 감싸지 않아도 됩니다. 대체할 수 있는 파라미터는 `%1`, `%2` 등으로 표시되지만, `%*` 는 모든 파라미터로 대체됩니다. 만약 명령이 주어지지 않으면, 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:

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.",k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

unalias name
지정된 에일리어스를 제거합니다.

! statement
현재 스택 프레임의 컨텍스트에서 단일 *statement*를 실행합니다. 문장의 첫 단어가 디버거 명령이 아닌 경우, 느낌표는 제외해도 됩니다. 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 인자가 주어진 경우, *shlex*으로 나뉘게 되고 결과는 새 *sys.argv*로 사용됩니다. 이전 기록, 중단점, 행동 그리고 디버거 옵션은 유지됩니다. *restart*는 *run*의 에일리어스입니다.

q(uit)
디버거를 종료합니다. 실행되고 있는 프로그램이 종료됩니다.

debug code
Enter a recursive debugger that steps through the code argument (which is an arbitrary expression or statement to be executed in the current environment).

retval
Print the return value for the last return of a function.

28.4 The Python Profilers

Source code: Lib/profile.py and Lib/pstats.py

28.4.1 Introduction to the profilers

cProfile and *profile* provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the *pstats* module.

The Python standard library provides two different implementations of the same profiling interface:

1. *cProfile* is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on lsprof, contributed by Brett Rosen and Ted Czotter.
2. *profile*, a pure Python module whose interface is imitated by *cProfile*, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

참고: The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is *timeit* for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

28.4.2 Instant User's Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use *profile* instead of *cProfile* if the latter is not available on your system.)

The above action would run *re.compile()* and print profile results like the following:

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

The first line indicates that 197 calls were monitored. Of those calls, 192 were *primitive*, meaning that the call was not induced via recursion. The next line: `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include:

ncalls for the number of calls.

tottime for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall is the quotient of `tottime` divided by `ncalls`

cumtime is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall is the quotient of `cumtime` divided by primitive calls

filename:lineno(function) provides the respective data of each function

When there are two numbers in the first column (for example `3/1`), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the `run()` function:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The *pstats.Stats* class reads profile results from a file and formats them in various ways.

The file *cProfile* can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

`-o` writes the profile results to a file instead of to stdout

`-s` specifies one of the *sort_stats()* sort values to sort the output by. This only applies when `-o` is not supplied.

`-m` specifies that a module is being profiled instead of a script.

버전 3.7에 추가: Added the `-m` option.

The *pstats* module's *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

28.4.3 `profile` and `cProfile` Module Reference

Both the *profile* and *cProfile* modules provide the following functions:

`profile.`**`run`**(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the *exec()* function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified, it is passed to this *Stats* instance to control how the results are sorted.

`profile.`**`runctx`**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to *run()*, with added arguments to supply the globals and locals dictionaries for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

`class` `profile.`**`Profile`**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

`enable`()

Start collecting profiling data.

`disable`()

Stop collecting profiling data.

create_stats()
Stop collecting profiling data and record the results internally as the current profile.

print_stats(*sort=-1*)
Create a *Stats* object based on the current profile and print the results to stdout.

dump_stats(*filename*)
Write the results of the current profile to *filename*.

run(*cmd*)
Profile the cmd via *exec()*.

runctx(*cmd*, *globals*, *locals*)
Profile the cmd via *exec()* with the specified global and local environment.

runcall(*func*, **args*, ***kwargs*)
Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a *sys.exit()* call during the called command/function execution) no profiling results will be printed.

28.4.4 The Stats Class

Analysis of the profiler data is done using the *Stats* class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)
This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or *profile.Profile* object can be used as the profile data source.

Stats objects have the following methods:

strip_dirs()
This method for the *Stats* class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If *strip_dirs()* causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)
This method of the *Stats* class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of *profile.run()* or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)
Save the data loaded into the *Stats* object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the *profile.Profile* and `cProfile.Profile` classes.

sort_stats(**keys*)

This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: `'time'`, `'name'`, `SortKey.TIME` or `SortKey.NAME`). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats(SortKey.NAME, SortKey.FILE)` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

Valid String Arg	Valid enum Arg	Meaning
`'calls'`	SortKey.CALLS	call count
`'cumulative'`	SortKey.CUMULATIVE	cumulative time
`'cumtime'`	N/A	cumulative time
`'file'`	N/A	file name
`'filename'`	SortKey.FILENAME	file name
`'module'`	N/A	file name
`'ncalls'`	N/A	call count
`'pcalls'`	SortKey.PCALLS	primitive call count
`'line'`	SortKey.LINE	line number
`'name'`	SortKey.NAME	function name
`'nfl'`	SortKey.NFL	name/file/line
`'stdname'`	SortKey.STDNAME	standard name
`'time'`	SortKey.TIME	internal time
`'tottime'`	N/A	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

버전 3.7에 추가: Added the SortKey enum.

reverse_order()

This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)

This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

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 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:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)

This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With *cProfile*, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)

This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

28.4.5 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, *statistical profiling* (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

28.4.6 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with `profile` than with the lower-overhead `cProfile`. For this reason, `profile` provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

28.4.7 Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see *Limitations*).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python's time.process_time() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

28.4.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *profile.Profile* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

profile.Profile `your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *os.times()* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see *Calibration*). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*os.times()* is *pretty* bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

cProfile.Profile `your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Python 3.3 adds several new functions in *time* that can be used to make precise measurements of process or wall-clock time. For example, see *time.perf_counter()*.

28.5 `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*에 있는 Tim Peters의 "Algorithms" 장의 개요도 참조하십시오.

28.5.1 기본 예제

다음 예제에서는 명령 줄 인터페이스를 사용하여 세 가지 다른 표현식을 비교하는 방법을 보여줍니다:

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 5: 27.5 usec per loop
$ python3 -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.

28.5.2 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()*입니다.

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 초 단위로 float로 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 None이 아니면, 각 시도 다음에 두 개의 인자로 호출합니다: `callback(number, time_taken)`.

버전 3.6에 추가.

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(...)       # outside the try/except
try:
    t.timeit(...)    # or 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.

28.5.3 명령 줄 인터페이스

명령 줄에서 프로그램으로 호출할 때, 다음 형식이 사용됩니다:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-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

  버전 3.3에 추가.

-u, **--unit**=U

  타이머 출력의 시간 단위를 지정합니다; nsec, usec, msec 또는 sec 중에서 선택할 수 있습니다.

  버전 3.5에 추가.

-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.

If *-n* is not given, a suitable number of loops is calculated by trying increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the total time is at least 0.2 seconds.

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 측정 할 수 있으며, 파이썬 버전마다 다를 수 있습니다.

28.5.4 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
```

```
>>> 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
>>> # attribute is missing
>>> 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
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     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():
    """Stupid test function"""
    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

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

28.6 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 명령줄에서 사용할 수 있습니다.

더 보기:

Coverage.py 브랜치 커버리지와 같은 고급 기능과 함께 HTML 출력을 제공하는 널리 사용되는 제삼자 커버리지 도구.

28.6.1 명령 줄 사용법

trace 모듈은 명령 줄에서 호출할 수 있습니다. 다음과 같이 간단할 수도 있습니다

```
python -m trace --count -C . somefile.py ...
```

이것은 somefile.py를 실행하고 실행 중에 임포트 한 모든 파이썬 모듈들의 주석이 달린 리스트를 현재 디렉터리에 생성합니다.

--help
사용법을 표시하고 종료합니다.

--version
모듈의 버전을 표시하고 종료합니다.

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.module에 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.

28.6.2 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*)
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.

프로그래밍 인터페이스의 사용을 보여주는 간단한 예제:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

28.7 tracemalloc — Trace memory allocations

버전 3.4에 추가.

Source code: Lib/tracemalloc.py

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`, or by using `-X tracemalloc` command line option. The *`tracemalloc.start()`* function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the `PYTHONTRACEMALLOC` environment variable to `25`, or use the `-X tracemalloc=25` command line option.

28.7.1 Examples

Display the top 10

Display the 10 files allocating the most memory:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite:

```
[ 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
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the *collections* module allocated `244 KiB` to build *namedtuple* types.

See *Snapshot.statistics()* for more options.

Compute differences

Take two snapshots and display the differences:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
/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
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the *linecache* module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the *Snapshot.dump()* method to analyze the snapshot offline. Then use the *Snapshot.load()* method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames):

```
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
  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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from modules: `870.1 KiB`. The traceback is where the *importlib* loaded data most recently: on the `import pdb` line of the *doctest* module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files:

```
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)

    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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite:

```
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
```

See *Snapshot.statistics()* for more options.

28.7.2 API

Functions

tracemalloc.**clear_traces**()
: Clear traces of memory blocks allocated by Python.

  See also *stop()*.

tracemalloc.**get_object_traceback**(*obj*)
: Get the traceback where the Python object *obj* was allocated. Return a *Traceback* instance, or `None` if the *tracemalloc* module is not tracing memory allocations or did not trace the allocation of the object.

  See also *gc.get_referrers()* and *sys.getsizeof()* functions.

tracemalloc.**get_traceback_limit**()
: Get the maximum number of frames stored in the traceback of a trace.

  The *tracemalloc* module must be tracing memory allocations to get the limit, otherwise an exception is raised.

  The limit is set by the *start()* function.

tracemalloc.**get_traced_memory**()
: Get the current size and peak size of memory blocks traced by the *tracemalloc* module as a tuple: `(current: int, peak: int)`.

tracemalloc.**get_tracemalloc_memory**()
Get the memory usage in bytes of the *tracemalloc* module used to store traces of memory blocks. Return an *int*.

tracemalloc.**is_tracing**()
`True` if the *tracemalloc* module is tracing Python memory allocations, `False` otherwise.

See also *start()* and *stop()* functions.

tracemalloc.**start**(*nframe: int=1*)
Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is `1`. *nframe* must be greater or equal to `1`.

Storing more than `1` frame is only useful to compute statistics grouped by `'traceback'` or to compute cumulative statistics: see the *Snapshot.compare_to()* and *Snapshot.statistics()* methods.

Storing more frames increases the memory and CPU overhead of the *tracemalloc* module. Use the *get_tracemalloc_memory()* function to measure how much memory is used by the *tracemalloc* module.

The `PYTHONTRACEMALLOC` environment variable (`PYTHONTRACEMALLOC=NFRAME`) and the `-X tracemalloc=NFRAME` command line option can be used to start tracing at startup.

See also *stop()*, *is_tracing()* and *get_traceback_limit()* functions.

tracemalloc.**stop**()
Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call *take_snapshot()* function to take a snapshot of traces before clearing them.

See also *start()*, *is_tracing()* and *clear_traces()* functions.

tracemalloc.**take_snapshot**()
Take a snapshot of traces of memory blocks allocated by Python. Return a new *Snapshot* instance.

The snapshot does not include memory blocks allocated before the *tracemalloc* module started to trace memory allocations.

Tracebacks of traces are limited to *get_traceback_limit()* frames. Use the *nframe* parameter of the *start()* function to store more frames.

The *tracemalloc* module must be tracing memory allocations to take a snapshot, see the *start()* function.

See also the *get_object_traceback()* function.

DomainFilter

class tracemalloc.**DomainFilter**(*inclusive: bool*, *domain: int*)
Filter traces of memory blocks by their address space (domain).

버전 3.6에 추가.

inclusive
If *inclusive* is `True` (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is `False` (exclude), match memory blocks not allocated in the address space *domain*.

domain
Address space of a memory block (`int`). Read-only property.

Filter

class tracemalloc.**Filter**(*inclusive: bool*, *filename_pattern: str*, *lineno: int=None*, *all_frames: bool=False*, *domain: int=None*)

Filter on traces of memory blocks.

See the *fnmatch.fnmatch()* function for the syntax of *filename_pattern*. The `'.pyc'` file extension is replaced with `'.py'`.

Examples:

- `Filter(True, subprocess.__file__)` only includes traces of the *subprocess* module
- `Filter(False, tracemalloc.__file__)` excludes traces of the *tracemalloc* module
- `Filter(False, "<unknown>")` excludes empty tracebacks

버전 3.5에서 변경: The `'.pyo'` file extension is no longer replaced with `'.py'`.

버전 3.6에서 변경: Added the *domain* attribute.

domain
Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive
If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno
Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern
Filename pattern of the filter (`str`). Read-only property.

all_frames
If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the *get_traceback_limit()* function and *Snapshot.traceback_limit* attribute.

Frame

class tracemalloc.**Frame**
Frame of a traceback.

The *Traceback* class is a sequence of *Frame* instances.

filename
Filename (`str`).

lineno
Line number (`int`).

Snapshot

class tracemalloc.**Snapshot**
Snapshot of traces of memory blocks allocated by Python.

The *take_snapshot()* function creates a snapshot instance.

compare_to(*old_snapshot: Snapshot*, *key_type: str*, *cumulative: bool=False*)
Compute the differences with an old snapshot. Get statistics as a sorted list of *StatisticDiff* instances grouped by *key_type*.

See the *Snapshot.statistics()* method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of *StatisticDiff.size_diff*, *StatisticDiff.size*, absolute value of *StatisticDiff.count_diff*, *Statistic.count* and then by *StatisticDiff.traceback*.

dump(*filename*)
Write the snapshot into a file.

Use *load()* to reload the snapshot.

filter_traces(*filters*)
Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *DomainFilter* and *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

버전 3.6에서 변경: *DomainFilter* instances are now also accepted in *filters*.

classmethod load(*filename*)
Load a snapshot from a file.

See also *dump()*.

statistics(*key_type: str*, *cumulative: bool=False*)
Get statistics as a sorted list of *Statistic* instances grouped by *key_type*:

key_type	description
`'filename'`	filename
`'lineno'`	filename and line number
`'traceback'`	traceback

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by: *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit
Maximum number of frames stored in the traceback of *traces*: result of the *get_traceback_limit()* when the snapshot was taken.

traces
Traces of all memory blocks allocated by Python: sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

Statistic

class tracemalloc.**Statistic**
Statistic on memory allocations.

Snapshot.statistics() returns a list of *Statistic* instances.

See also the *StatisticDiff* class.

count
Number of memory blocks (int).

size
Total size of memory blocks in bytes (int).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

StatisticDiff

class tracemalloc.**StatisticDiff**
Statistic difference on memory allocations between an old and a new *Snapshot* instance.

Snapshot.compare_to() returns a list of *StatisticDiff* instances. See also the *Statistic* class.

count
Number of memory blocks in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

count_diff
Difference of number of memory blocks between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

size
Total size of memory blocks in bytes in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

size_diff
Difference of total size of memory blocks in bytes between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

traceback
Traceback where the memory blocks were allocated, *Traceback* instance.

Trace

class tracemalloc.**Trace**
Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

버전 3.6에서 변경: Added the *domain* attribute.

domain
Address space of a memory block (int). Read-only property.

tracemalloc uses the domain 0 to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size
Size of the memory block in bytes (`int`).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

Traceback

class tracemalloc.**Traceback**
Sequence of *Frame* instances sorted from the oldest frame to the most recent frame.

A traceback contains at least `1` frame. If the `tracemalloc` module failed to get a frame, the filename `"<unknown>"` at line number `0` is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function.

The *Trace.traceback* attribute is an instance of *Traceback* instance.

버전 3.7에서 변경: Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

format(*limit=None*, *most_recent_first=False*)
Format the traceback as a list of lines with newlines. Use the *linecache* module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the `abs(limit)` oldest frames. If *most_recent_first* is `True`, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the *traceback.format_tb()* function, except that *format()* does not include newlines.

Example:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Output:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPTER 29

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.

29.1 distutils — 파이썬 모듈 빌드와 설치

distutils 패키지는 파이썬 설치에 추가 모듈을 빌드하고 설치하는 것을 지원합니다. 새 모듈은 100% 순수 파이썬이거나 C로 작성된 확장 모듈일 수도 있고, 파이썬과 C로 코딩된 모듈을 포함하는 파이썬 패키지 모음일 수도 있습니다.

대부분 파이썬 사용자는 직접 이 모듈을 사용하려고 하지 않을겁니다, 대신 파이썬 패키징 위원회가 유지하는 교차 버전 도구를 사용합니다. 특히, setuptools는 다음을 제공하는 *distutils*의 향상된 대안입니다:

- 프로젝트 의존성 선언 지원
- 소스 배포에 포함할 파일을 구성하기 위한 추가 메커니즘 (버전 제어 시스템과의 통합을 위한 플러그인 포함)
- 응용 프로그램 플러그인 시스템의 기초로 사용할 수 있는 프로젝트 "진입점"을 선언할 수 있는 능력
- 미리 빌드 할 필요 없이, 설치 시 윈도우 명령 줄 실행 파일을 자동으로 생성하는 능력
- 지원되는 모든 파이썬 버전에서 일관된 동작

권장되는 pip 설치 관리자는 `setuptools`로 모든 `setup.py` 스크립트를 실행합니다. 스크립트 자체가 `distutils` 만 임포트 할 때조차 그렇습니다. 자세한 내용은 파이썬 패키징 사용자 지침서를 참조하십시오.

패키징 도구 작성자와 현재 패키징과 배포 시스템의 세부 사항을 더 깊이 이해하고자 하는 사용자를 위해, 기존의 *distutils* 기반 사용자 설명서와 API 레퍼런스를 계속 제공합니다:

- install-index
- distutils-index

29.2 ensurepip — pip 설치 프로그램 부트스트랩

버전 3.4에 추가.

---

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.

29.2.1 명령 줄 인터페이스

명령 줄 인터페이스는 인터프리터의 -m 스위치를 사용하여 호출됩니다.

가장 간단한 호출은 이렇습니다:

```
python -m ensurepip
```

이 호출은 아직 설치되지 않았으면 pip를 설치하지만, 그렇지 않으면 아무것도 하지 않습니다. pip의 설치 버전이 적어도 ensurepip에 번들 된 최신 버전이 되도록 하려면, --upgrade 옵션을 전달하십시오:

```
python -m ensurepip --upgrade
```

기본적으로, pip는 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 pip를 설치합니다.
- --user: pip를 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 ensurepip를 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: if a "default pip" installation is requested, the pip script will be installed in addition to the two regular scripts.

두 스크립트 선택 옵션을 모두 제공하면 예외가 발생합니다.

29.2.2 모듈 API

*ensurepip*는 프로그래밍 방식으로 사용하기 위해 두 가지 함수를 제공합니다:

ensurepip.**version**()
환경을 부트스트랩 할 때 설치될 pip의 번들 버전을 지정하는 문자열을 반환합니다.

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)
pip를 현재나 지정된 환경으로 부트스트랩 합니다.

*root*는 상대 경로로 설치할 대안 루트 디렉터리를 지정합니다. *root*가 None이면, 설치는 현재 환경의 기본 설치 위치를 사용합니다.

*upgrade*는 이미 설치된 이전 버전의 pip를 번들 된 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, pipX가 설치되지 않습니다.

*default_pip*가 설정되면, 두 개의 일반 스크립트에 더해 pip가 설치됩니다.

altinstall 과 *default_pip*를 모두 설정하면 *ValueError*가 발생합니다.

*verbosity*는 부트스트랩 연산에서 *sys.stdout*로 출력하는 수준을 제어합니다.

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 pip에 필요한 추가 모듈을 설치할 수 있지만, 다른 소프트웨어는 이러한 종속성이 기본적으로 항상 존재한다고 가정해서는 안 됩니다 (pip의 차후 버전에서 제거될 수 있기 때문입니다).

29.3 venv — 가상 환경 생성

버전 3.3에 추가.

소스 코드: Lib/venv/

---

venv 모듈은 자체 사이트 디렉터리를 갖는 경량 "가상 환경"을 만들고, 선택적으로 시스템 사이트 디렉터리에서 격리할 수 있도록 지원합니다. 각 가상 환경은 고유한 파이썬 바이너리 (이 환경을 만드는 데 사용된 바이너리 버전과 일치함)를 가지며 자신의 사이트 디렉터리에 독립적으로 설치된 파이썬 패키지 집합을 가질 수 있습니다.

파이썬 가상 환경에 대한 자세한 내용은 **PEP 405**를 참조하십시오.

더 보기:

Python Packaging User Guide: Creating and using virtual environments

---

참고: pyvenv 스크립트는 파이썬 3.6 에서 폐지되었고, 가상 환경이 어떤 파이썬 인터프리터를 기반으로 하는지에 대한 잠재적인 혼동을 방지하기 위해 python3 -m venv를 사용합니다.

29.3.1 가상 환경 만들기

가상 환경은 venv 명령을 실행해서 만들어집니다:

```
python3 -m venv /path/to/new/virtual/environment
```

이 명령을 실행하면 대상 디렉터리가 만들어지고 (이미 존재하지 않는 부모 디렉터리도 만듭니다) 명령이 실행된 파이썬 설치를 가리키는 home 키가 있는 pyvenv.cfg 파일이 배치됩니다. 또한 파이썬 바이너리/바이너리들의 사본/심볼릭 링크를 포함하는 bin (또는 윈도우의 경우 Scripts) 서브 디렉터리를 만듭니다 (플랫폼이나 환경 생성 시에 사용된 인자에 적절하게). 또한 (처음에는 비어있는) lib/pythonX.Y/site-packages (윈도우에서는, Lib\site-packages) 서브 디렉터리를 만듭니다. 기존 디렉터리가 지정되면 재사용됩니다.

버전 3.6부터 폐지: pyvenv는 파이썬 3.3 및 3.4 용 가상 환경을 만드는 데 권장되는 도구였으며, 파이썬 3.6에서 폐지되었습니다.

버전 3.5에서 변경: 이제 가상 환경을 만들 때 venv를 사용하는 것이 좋습니다.

윈도우에서는, 다음과 같이 venv 명령을 호출하십시오:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

또는, 여러분의 파이썬 설치에 PATH 와 PATHEXT 변수를 구성했으면:

```
c:\>python -m venv c:\path\to\myenv
```

명령에 -h를 사용하면 사용 가능한 옵션이 표시됩니다:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --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.

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

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.

참고: 심볼릭 링크가 윈도우에서 지원되지만, 추천하지는 않습니다. 특히 파일 탐색기에서 `python.exe`를 더블 클릭하면 심볼릭 링크를 열심히 따라가고(resolve) 가상 환경은 무시됩니다.

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.

일단 가상 환경이 만들어지면, 가상 환경의 바이너리 디렉터리에 있는 스크립트를 사용하여 "활성화"할 수 있습니다. 스크립트의 호출은 플랫폼에 따라 다릅니다 (<*venv*>는 가상 환경을 포함하는 디렉터리의 경로로 대체되어야 합니다):

플랫폼	셸	가상 환경을 활성화하는 명령
Posix	bash/zsh	$ source <venv>/bin/activate
	fish	$. <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
윈도우	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

환경을 구체적으로 활성화할 필요는 없습니다; 활성화는 단지 PATH의 처음에 가상 환경의 바이너리 디렉터리를 추가해서, "python"이 가상 환경의 파이썬 인터프리터를 호출하고 전체 경로를 사용하지 않고도 설치된 스크립트를 실행할 수 있도록 할 뿐입니다. 그러나, 가상 환경에 설치된 모든 스크립트는 활성화하지 않고도 실행 가능해야 하며, 자동으로 가상 환경의 파이썬을 사용하여 실행해야 합니다.

You can deactivate a virtual environment by typing "deactivate" in your shell. The exact mechanism is platform-specific and is an internal implementation detail (typically a script or shell function will be used).

버전 3.4에 추가: `fish` 와 `csh` 활성화 스크립트.

참고: 가상 환경은 파이썬 인터프리터, 라이브러리 및 스크립트가 다른 가상 환경에 설치된 것과 (기본적으로) "시스템" 파이썬 (즉, 여러분의 운영 체제 일부로 설치되어있는 것)에 설치된 모든 라이브러리와 격리되어있는 파이썬 환경입니다.

가상 환경은 파이썬 실행 파일과 가상 환경임을 나타내는 다른 파일을 포함하는 디렉터리 트리입니다.

Common installation tools such as setuptools and pip work as expected with virtual environments. In other words, when a virtual environment is active, they install Python packages into the virtual environment without needing to be told to do so explicitly.

가상 환경이 활성일 때 (즉, 가상 환경의 파이썬 인터프리터가 실행 중일 때), 어트리뷰트 *sys.prefix* 와 *sys.exec_prefix*는 가상 환경의 베이스 디렉터리를 가리키지만, *sys.base_prefix*와 *sys.base_exec_prefix*는 가상 환경을 만들 때 사용한 가상이 아닌 환경의 파이썬을 가리킵니다. 가상 환경이 활성화되어 있지 않으면, *sys.prefix*는 *sys.base_prefix*와 같고, *sys.exec_prefix*는 *sys.base_exec_prefix*와 같습니다 (모두 가상 환경이 아닌 파이썬 설치를 가리킵니다).

When a virtual environment is active, any options that change the installation path will be ignored from all *distutils* configuration files to prevent projects being inadvertently installed outside of the virtual environment.

When working in a command shell, users can make a virtual environment active by running an `activate` script in the virtual environment's executables directory (the precise filename and command to use the file is shell-dependent),

which prepends the virtual environment's directory for executables to the PATH environment variable for the running shell. There should be no need in other circumstances to activate a virtual environment; scripts installed into virtual environments have a "shebang" line which points to the virtual environment's Python interpreter. 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 Python Launcher for Windows installed (this was added to Python in 3.3 - see **PEP 397** for more details). Thus, double-clicking an installed script in a Windows Explorer window should run the script with the correct interpreter without there needing to be any reference to its virtual environment in PATH.

29.3.2 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*)

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 None입니다).

버전 3.4에서 변경: with_pip 매개 변수 추가

버전 3.6에 추가: prompt 매개 변수 추가

Creators of third-party virtual environment tools will be free to use the provided *EnvBuilder* class as a base class.

반환된 객체에는 메서드 create가 있습니다:

create(*env_dir*)

Create a virtual environment by specifying the target directory (absolute or relative to the current directory) which is to contain the virtual environment. The create method will either create the environment in the specified directory, or raise an appropriate exception.

The create method of the *EnvBuilder* class illustrates the hooks available for subclass customization:

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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 필요한 모든 디렉터리를 만들고 문맥 객체를 돌려줍니다. 이것은 다른 메서드에서 사용하기 위한 어트리뷰트(가령 경로)를 담고 있을 뿐입니다. 기존 환경 디렉터리에 작동하도록 `clear` 나 `upgrade`가 지정되어있는 한, 디렉터리는 이미 존재할 수 있습니다.

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.

post_setup(*context*)
제삼자 구현에서 재정의하여 가상 환경에 패키지를 사전 설치하거나 다른 생성 후 단계를 수행할 수 있는 메서드입니다.

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.

또한, *EnvBuilder*는 가상 환경에 사용자 정의 스크립트를 설치하는 데 도움이 되는 유틸리티 메서드를 제공하는데, 서브 클래스의 *setup_scripts()* 나 *post_setup()*에서 호출할 수 있습니다.

install_scripts(*context*, *path*)
*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).

모듈 수준의 편리 함수도 있습니다:

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*)
주어진 키워드 인자로 *EnvBuilder*를 만들고, *env_dir* 인자로 *create()* 메서드를 호출합니다.

버전 3.3에 추가.

버전 3.4에서 변경: with_pip 매개 변수 추가

버전 3.6에서 변경: prompt 매개 변수 추가

29.3.3 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):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
                     The context argument can have one of three values:
                     'main', indicating that it is called from virtualize()
                     itself, and 'stdout' and 'stderr', which are obtained
                     by reading lines from the output streams of a subprocess
                     which is used to install the app.

                     If a callable is not specified, default progress
                     information is output to sys.stderr.
    """

    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):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        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)
        # Download script into the virtual environment's binaries folder
        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()
        # Install in the virtual environment
        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()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bitbucket.org/pypa/setuptools/downloads/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://raw.github.com/pypa/pip/master/contrib/get-pip.py'
        self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                         'Python 3.3 or later')
    else:
        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,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                            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 '
                                 '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.

29.4 zipapp — Manage executable Python zip archives

버전 3.5에 추가.

Source code: Lib/zipapp.py

This module provides tools to manage the creation of zip files containing Python code, which can be executed directly by the Python interpreter. The module provides both a *Command-Line Interface* and a *Python API*.

29.4.1 Basic Example

The following example shows how the *Command-Line Interface* can be used to create an executable archive from a directory containing Python code. When run, the archive will execute the `main` function from the module `myapp` in the archive.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

29.4.2 Command-Line Interface

When called as a program from the command line, the following form is used:

```
$ python -m zipapp source [options]
```

If *source* is a directory, this will create an archive from the contents of *source*. If *source* is a file, it should be an archive, and it will be copied to the target archive (or the contents of its shebang line will be displayed if the –info option is specified).

The following options are understood:

-o <output>, **--output**=<output>
: Write the output to a file named *output*. If this option is not specified, the output filename will be the same as the input *source*, with the extension `.pyz` added. If an explicit filename is given, it is used as is (so a `.pyz` extension should be included if required).

  An output filename must be specified if the *source* is an archive (and in that case, *output* must not be the same as *source*).

-p <interpreter>, **--python**=<interpreter>
: Add a `#!` line to the archive specifying *interpreter* as the command to run. Also, on POSIX, make the archive executable. The default is to write no `#!` line, and not make the file executable.

-m <mainfn>, **--main**=<mainfn>
: Write a `__main__.py` file to the archive that executes *mainfn*. The *mainfn* argument should have the form "pkg.mod:fn", where "pkg.mod" is a package/module in the archive, and "fn" is a callable in the given module. The `__main__.py` file will execute that callable.

  --main cannot be specified when copying an archive.

-c, --compress
: Compress files with the deflate method, reducing the size of the output file. By default, files are stored uncompressed in the archive.

  --compress has no effect when copying an archive.

  버전 3.7에 추가.

--info
Display the interpreter embedded in the archive, for diagnostic purposes. In this case, any other options are ignored and SOURCE must be an archive, not a directory.

-h, --help
Print a short usage message and exit.

29.4.3 Python API

The module defines two convenience functions:

zipapp.**create_archive**(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)
Create an application archive from *source*. The source can be any of the following:

- The name of a directory, or a *path-like object* referring to a directory, in which case a new application archive will be created from the content of that directory.
- The name of an existing application archive file, or a *path-like object* referring to such a file, in which case the file is copied to the target (modifying it to reflect the value given for the *interpreter* argument). The file name should include the `.pyz` extension, if required.
- A file object open for reading in bytes mode. The content of the file should be an application archive, and the file object is assumed to be positioned at the start of the archive.

The *target* argument determines where the resulting archive will be written:

- If it is the name of a file, or a *path-like object*, the archive will be written to that file.
- If it is an open file object, the archive will be written to that file object, which must be open for writing in bytes mode.
- If the target is omitted (or `None`), the source must be a directory and the target will be a file with the same name as the source, with a `.pyz` extension added.

The *interpreter* argument specifies the name of the Python interpreter with which the archive will be executed. It is written as a "shebang" line at the start of the archive. On POSIX, this will be interpreted by the OS, and on Windows it will be handled by the Python launcher. Omitting the *interpreter* results in no shebang line being written. If an interpreter is specified, and the target is a filename, the executable bit of the target file will be set.

The *main* argument specifies the name of a callable which will be used as the main program for the archive. It can only be specified if the source is a directory, and the source does not already contain a `__main__.py` file. The *main* argument should take the form "pkg.module:callable" and the archive will be run by importing "pkg.module" and executing the given callable with no arguments. It is an error to omit *main* if the source is a directory and does not contain a `__main__.py` file, as otherwise the resulting archive would not be executable.

The optional *filter* argument specifies a callback function that is passed a Path object representing the path to the file being added (relative to the source directory). It should return `True` if the file is to be added.

The optional *compressed* argument determines whether files are compressed. If set to `True`, files in the archive are compressed with the deflate method; otherwise, files are stored uncompressed. This argument has no effect when copying an existing archive.

If a file object is specified for *source* or *target*, it is the caller's responsibility to close it after calling create_archive.

When copying an existing archive, file objects supplied only need `read` and `readline`, or `write` methods. When creating an archive from a directory, if the target is a file object it will be passed to the `zipfile.ZipFile` class, and must supply the methods needed by that class.

버전 3.7에 추가: Added the *filter* and *compressed* arguments.

zipapp.**get_interpreter**(*archive*)
Return the interpreter specified in the `#!` line at the start of the archive. If there is no `#!` line, return *None*. The *archive* argument can be a filename or a file-like object open for reading in bytes mode. It is assumed to be at the start of the archive.

29.4.4 Examples

Pack up a directory into an archive, and run it.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

The same can be done using the *create_archive()* function:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

To make the application directly executable on POSIX, specify an interpreter to use.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

To replace the shebang line on an existing archive, create a modified archive using the *create_archive()* function:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

To update the file in place, do the replacement in memory using a `BytesIO` object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file. This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory:

```
>>> 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())
```

29.4.5 Specifying the Interpreter

Note that if you specify an interpreter and then distribute your application archive, you need to ensure that the interpreter used is portable. The Python launcher for Windows supports most common forms of POSIX `#!` line, but there are other issues to consider:

- If you use "/usr/bin/env python" (or other forms of the "python" command, such as "/usr/bin/python"), you need to consider that your users may have either Python 2 or Python 3 as their default, and write your code to work under both versions.
- If you use an explicit version, for example "/usr/bin/env python3" your application will not work for users who do not have that version. (This may be what you want if you have not made your code Python 2 compatible).
- There is no way to say "python X.Y or later", so be careful of using an exact version like "/usr/bin/env python3.4" as you will need to change your shebang line for users of Python 3.5, for example.

Typically, you should use an "/usr/bin/env python2" or "/usr/bin/env python3", depending on whether your code is written for Python 2 or 3.

29.4.6 Creating Standalone Applications with zipapp

Using the *zipapp* module, it is possible to create self-contained Python programs, which can be distributed to end users who only need to have a suitable version of Python installed on their system. The key to doing this is to bundle all of the application's dependencies into the archive, along with the application code.

The steps to create a standalone archive are as follows:

1. Create your application in a directory as normal, so you have a `myapp` directory containing a `__main__.py` file, and any supporting application code.
2. Install all of your application's dependencies into the `myapp` directory, using pip:

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   (this assumes you have your project requirements in a `requirements.txt` file - if not, you can just list the dependencies manually on the pip command line).
3. Optionally, delete the `.dist-info` directories created by pip in the `myapp` directory. These hold metadata for pip to manage the packages, and as you won't be making any further use of pip they aren't required - although it won't do any harm if you leave them.
4. Package the application using:

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

This will produce a standalone executable, which can be run on any machine with the appropriate interpreter available. See *Specifying the Interpreter* for details. It can be shipped to users as a single file.

On Unix, the `myapp.pyz` file is executable as it stands. You can rename the file to remove the `.pyz` extension if you prefer a "plain" command name. On Windows, the `myapp.pyz[w]` file is executable by virtue of the fact that the Python interpreter registers the `.pyz` and `.pyzw` file extensions when installed.

Making a Windows executable

On Windows, registration of the `.pyz` extension is optional, and furthermore, there are certain places that don't recognise registered extensions "transparently" (the simplest example is that `subprocess.run(['myapp'])` won't find your application - you need to explicitly specify the extension).

On Windows, therefore, it is often preferable to create an executable from the zipapp. This is relatively easy, although it does require a C compiler. The basic approach relies on the fact that zipfiles can have arbitrary data prepended, and Windows exe files can have arbitrary data appended. So by creating a suitable launcher and tacking the `.pyz` file onto the end of it, you end up with a single-file executable that runs your application.

A suitable launcher can be as simple as the following:

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
#else
int wmain()
#endif
{
    wchar_t **myargv = _alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}
```

If you define the `WINDOWS` preprocessor symbol, this will generate a GUI executable, and without it, a console executable.

To compile the executable, you can either just use the standard MSVC command line tools, or you can take advantage of the fact that distutils knows how to compile Python source:

```
>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe + 'w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")
```

The resulting launcher uses the “Limited ABI”, so it will run unchanged with any version of Python 3.x. All it needs is for Python (`python3.dll`) to be on the user’s `PATH`.

For a fully standalone distribution, you can distribute the launcher with your application appended, bundled with the Python “embedded” distribution. This will run on any PC with the appropriate architecture (32 bit or 64 bit).

Caveats

There are some limitations to the process of bundling your application into a single file. In most, if not all, cases they can be addressed without needing major changes to your application.

1. If your application depends on a package that includes a C extension, that package cannot be run from a zip file (this is an OS limitation, as executable code must be present in the filesystem for the OS loader to load it). In this case, you can exclude that dependency from the zipfile, and either require your users to have it installed, or ship it alongside your zipfile and add code to your `__main__.py` to include the directory containing the unzipped module in `sys.path`. In this case, you will need to make sure to ship appropriate binaries for your target architecture(s) (and potentially pick the correct version to add to `sys.path` at runtime, based on the user's machine).
2. If you are shipping a Windows executable as described above, you either need to ensure that your users have `python3.dll` on their PATH (which is not the default behaviour of the installer) or you should bundle your application with the embedded distribution.
3. The suggested launcher above uses the Python embedding API. This means that in your application, `sys.executable` will be your application, and *not* a conventional Python interpreter. Your code and its dependencies need to be prepared for this possibility. For example, if your application uses the *multiprocessing* module, it will need to call *multiprocessing.set_executable()* to let the module know where to find the standard Python interpreter.

29.4.7 The Python Zip Application Archive Format

Python has been able to execute zip files which contain a `__main__.py` file since version 2.6. In order to be executed by Python, an application archive simply has to be a standard zip file containing a `__main__.py` file which will be run as the entry point for the application. As usual for any Python script, the parent of the script (in this case the zip file) will be placed on *sys.path* and thus further modules can be imported from the zip file.

The zip file format allows arbitrary data to be prepended to a zip file. The zip application format uses this ability to prepend a standard POSIX "shebang" line to the file (`#!/path/to/interpreter`).

Formally, the Python zip application format is therefore:

1. An optional shebang line, containing the characters `b'#!'` followed by an interpreter name, and then a newline (`b'\n'`) character. The interpreter name can be anything acceptable to the OS "shebang" processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in *sys.getfilesystemencoding()* on POSIX.
2. Standard zipfile data, as generated by the *zipfile* module. The zipfile content *must* include a file called `__main__.py` (which must be in the "root" of the zipfile - i.e., it cannot be in a subdirectory). The zipfile data can be compressed or uncompressed.

If an application archive has a shebang line, it may have the executable bit set on POSIX systems, to allow it to be executed directly.

There is no requirement that the tools in this module are used to create application archives - the module is a convenience, but archives in the above format created by any means are acceptable to Python.

CHAPTER 30

파이썬 실행시간 서비스

이 장에서 설명하는 모듈들은 파이썬 인터프리터와 그 환경과의 상호 작용과 관련된 다양한 서비스를 제공합니다. 다음은 개요입니다:

30.1 sys — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

sys.**abiflags**
On POSIX systems where Python was built with the standard `configure` script, this contains the ABI flags as specified by **PEP 3149**.

버전 3.2에 추가.

sys.**argv**
The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `'-c'`. If no script name was passed to the Python interpreter, `argv[0]` is the empty string.

To loop over the standard input, or the list of files given on the command line, see the *fileinput* module.

참고: On Unix, command line arguments are passed by bytes from OS. Python decodes them with filesystem encoding and "surrogateescape" error handler. When you need original bytes, you can get it by `[os.fsencode(arg) for arg in sys.argv]`.

---

sys.**base_exec_prefix**
Set during Python startup, before `site.py` is run, to the same value as *exec_prefix*. If not running in a *virtual environment*, the values will stay the same; if `site.py` finds that a virtual environment is in use, the values of *prefix* and *exec_prefix* will be changed to point to the virtual environment, whereas *base_prefix* and

base_exec_prefix will remain pointing to the base Python installation (the one which the virtual environment was created from).

버전 3.3에 추가.

sys.**base_prefix**
Set during Python startup, before site.py is run, to the same value as *prefix*. If not running in a *virtual environment*, the values will stay the same; if site.py finds that a virtual environment is in use, the values of *prefix* and *exec_prefix* will be changed to point to the virtual environment, whereas *base_prefix* and *base_exec_prefix* will remain pointing to the base Python installation (the one which the virtual environment was created from).

버전 3.3에 추가.

sys.**byteorder**
An indicator of the native byte order. This will have the value 'big' on big-endian (most-significant byte first) platforms, and 'little' on little-endian (least-significant byte first) platforms.

sys.**builtin_module_names**
A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — modules.keys() only lists the imported modules.)

sys.**call_tracing**(*func*, *args*)
Call func(*args), while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug some other code.

sys.**copyright**
A string containing the copyright pertaining to the Python interpreter.

sys.**_clear_type_cache**()
Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function *only* to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

sys.**_current_frames**()
Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the *traceback* module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

sys.**breakpointhook**()
This hook function is called by built-in *breakpoint()*. By default, it drops you into the *pdb* debugger, but it can be set to any other function so that you can choose which debugger gets used.

The signature of this function is dependent on what it calls. For example, the default binding (e.g. pdb.set_trace()) expects no arguments, but you might bind it to a function that expects additional arguments (positional and/or keyword). The built-in breakpoint() function passes its *args and **kws straight through. Whatever breakpointhooks() returns is returned from breakpoint().

The default implementation first consults the environment variable PYTHONBREAKPOINT. If that is set to "0" then this function returns immediately; i.e. it is a no-op. If the environment variable is not set, or is set to the empty string, pdb.set_trace() is called. Otherwise this variable should name a function to run, using Python's dotted-import nomenclature, e.g. package.subpackage.module.function. In this case, package.subpackage.module would be imported and the resulting module must have a callable named function().

This is run, passing in `*args` and `**kws`, and whatever `function()` returns, `sys.breakpointhook()` returns to the built-in *`breakpoint()`* function.

Note that if anything goes wrong while importing the callable named by `PYTHONBREAKPOINT`, a *`RuntimeWarning`* is reported and the breakpoint is ignored.

Also note that if `sys.breakpointhook()` is overridden programmatically, `PYTHONBREAKPOINT` is *not* consulted.

버전 3.7에 추가.

`sys.`**`_debugmallocstats`**`()`
Print low-level information to stderr about the state of CPython's memory allocator.

If Python is configured –with-pydebug, it also performs some expensive internal consistency checks.

버전 3.3에 추가.

CPython implementation detail: This function is specific to CPython. The exact output format is not defined here, and may change.

`sys.`**`dllhandle`**
Integer specifying the handle of the Python DLL.

Availability: Windows.

`sys.`**`displayhook`**`(`*value*`)`
If *value* is not `None`, this function prints `repr(value)` to `sys.stdout`, and saves *value* in `builtins._`. If `repr(value)` is not encodable to `sys.stdout.encoding` with `sys.stdout.errors` error handler (which is probably `'strict'`), encode it to `sys.stdout.encoding` with `'backslashreplace'` error handler.

`sys.displayhook` is called on the result of evaluating an *expression* entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

Pseudo-code:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        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: Use `'backslashreplace'` error handler on *`UnicodeEncodeError`*.

`sys.`**`dont_write_bytecode`**
If this is true, Python won't try to write `.pyc` files on the import of source modules. This value is initially set to `True` or `False` depending on the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable, but you can set it yourself to control bytecode file generation.

sys.**excepthook**(*type*, *value*, *traceback*)
This function prints out a given traceback and exception to sys.stderr.

When an exception is raised and uncaught, the interpreter calls sys.excepthook with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to sys.excepthook.

sys.**__breakpointhook__**
sys.**__displayhook__**
sys.**__excepthook__**
These objects contain the original values of breakpointhook, displayhook, and excepthook at the start of the program. They are saved so that breakpointhook, displayhook and excepthook can be restored in case they happen to get replaced with broken or alternative objects.

버전 3.7에 추가: __breakpointhook__

sys.**exc_info**()
This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, "handling an exception" is defined as "executing an except clause." For any stack frame, only information about the exception being currently handled is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three None values is returned. Otherwise, the values returned are (type, value, traceback). Their meaning is: *type* gets the type of the exception being handled (a subclass of *BaseException*); *value* gets the exception instance (an instance of the exception type); *traceback* gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

sys.**exec_prefix**
A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '/usr/local'. This can be set at build time with the --exec-prefix argument to the **configure** script. Specifically, all configuration files (e.g. the pyconfig.h header file) are installed in the directory *exec_prefix*/lib/python*X.Y*/config, and shared library modules are installed in *exec_prefix*/lib/python*X.Y*/lib-dynload, where *X.Y* is the version number of Python, for example 3.2.

참고: If a *virtual environment* is in effect, this value will be changed in site.py to point to the virtual environment. The value for the Python installation will still be available, via *base_exec_prefix*.

---

sys.**executable**
A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, *sys.executable* will be an empty string or None.

sys.**exit**([*arg*])
Exit from Python. This is implemented by raising the *SystemExit* exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level.

The optional argument *arg* can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, None is equivalent to passing zero, and any other object is printed

to *stderr* and results in an exit code of 1. In particular, `sys.exit("some error message")` is a quick way to exit a program when an error occurs.

Since *exit()* ultimately "only" raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted.

버전 3.6에서 변경: If an error occurs in the cleanup after the Python interpreter has caught *SystemExit* (such as an error flushing buffered data in the standard streams), the exit status is changed to 120.

`sys.`**`flags`**

The *named tuple* *flags* exposes the status of command line flags. The attributes are read only.

attribute	flag
`debug`	`-d`
`inspect`	`-i`
`interactive`	`-i`
`isolated`	`-I`
`optimize`	`-O` or `-OO`
`dont_write_bytecode`	`-B`
`no_user_site`	`-s`
`no_site`	`-S`
`ignore_environment`	`-E`
`verbose`	`-v`
`bytes_warning`	`-b`
`quiet`	`-q`
`hash_randomization`	`-R`
`dev_mode`	`-X dev`
`utf8_mode`	`-X utf8`
`int_max_str_digits`	`-X int_max_str_digits` (*integer string conversion length limitation*)

버전 3.2에서 변경: Added `quiet` attribute for the new `-q` flag.

버전 3.2.3에 추가: The `hash_randomization` attribute.

버전 3.3에서 변경: Removed obsolete `division_warning` attribute.

버전 3.4에서 변경: Added `isolated` attribute for `-I isolated` flag.

버전 3.7에서 변경: Added `dev_mode` attribute for the new `-X dev` flag and `utf8_mode` attribute for the new `-X utf8` flag.

버전 3.7.14에서 변경: Added the `int_max_str_digits` attribute.

`sys.`**`float_info`**

A *named tuple* holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the 'C' programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], 'Characteristics of floating types', for details.

attribute	float.h macro	explanation
epsilon	DBL_EPSILON	difference between 1.0 and the least value greater than 1.0 that is representable as a float
dig	DBL_DIG	maximum number of decimal digits that can be faithfully represented in a float; see below
mant_dig	DBL_MANT_DIG	float precision: the number of base-radix digits in the significand of a float
max	DBL_MAX	maximum representable positive finite float
max_exp	DBL_MAX_EXP	maximum integer *e* such that radix**(e-1) is a representable finite float
max_10_exp	DBL_MAX_10_EXP	maximum integer *e* such that 10**e is in the range of representable finite floats
min	DBL_MIN	minimum representable positive *normalized* float
min_exp	DBL_MIN_EXP	minimum integer *e* such that radix**(e-1) is a normalized float
min_10_exp	DBL_MIN_10_EXP	minimum integer *e* such that 10**e is a normalized float
radix	FLT_RADIX	radix of exponent representation
rounds	FLT_ROUNDS	integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.

The attribute sys.float_info.dig needs further explanation. If s is any string representing a decimal number with at most sys.float_info.dig significant digits, then converting s to a float and back again will recover a string representing the same decimal value:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

But for strings with more than sys.float_info.dig significant digits, this isn't always true:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.**float_repr_style**

A string indicating how the *repr()* function behaves for floats. If the string has value 'short' then for a finite float x, repr(x) aims to produce a short string with the property that float(repr(x)) == x. This is the usual behaviour in Python 3.1 and later. Otherwise, float_repr_style has value 'legacy' and repr(x) behaves in the same way as it did in versions of Python prior to 3.1.

버전 3.1에 추가.

sys.**getallocatedblocks**()

Return the number of memory blocks currently allocated by the interpreter, regardless of their size. This function is mainly useful for tracking and debugging memory leaks. Because of the interpreter's internal caches, the result can vary from call to call; you may have to call *_clear_type_cache()* and *gc.collect()* to get more predictable results.

If a Python build or implementation cannot reasonably compute this information, *getallocatedblocks()*

is allowed to return 0 instead.

버전 3.4에 추가.

sys.**getandroidapilevel**()
Return the build time API version of Android as an integer.

Availability: Android.

버전 3.7에 추가.

sys.**getcheckinterval**()
Return the interpreter's "check interval"; see *sys.setcheckinterval()*.

버전 3.2부터 폐지: Use *sys.getswitchinterval()* instead.

sys.**getdefaultencoding**()
Return the name of the current default string encoding used by the Unicode implementation.

sys.**getdlopenflags**()
Return the current value of the flags that are used for `dlopen()` calls. Symbolic names for the flag values can be found in the *os* module (`RTLD_xxx` constants, e.g. *os.RTLD_LAZY*).

Availability: Unix.

sys.**getfilesystemencoding**()
Return the name of the encoding used to convert between Unicode filenames and bytes filenames. For best compatibility, str should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either str or bytes and internally convert to the system's preferred representation.

This encoding is always ASCII-compatible.

os.fsencode() and *os.fsdecode()* should be used to ensure that the correct encoding and errors mode are used.

- In the UTF-8 mode, the encoding is `utf-8` on any platform.
- On Mac OS X, the encoding is `'utf-8'`.
- On Unix, the encoding is the locale encoding.
- On Windows, the encoding may be `'utf-8'` or `'mbcs'`, depending on user configuration.

버전 3.2에서 변경: *sys.getfilesystemencoding()* result cannot be `None` anymore.

버전 3.6에서 변경: Windows is no longer guaranteed to return `'mbcs'`. See **PEP 529** and *sys._enablelegacywindowsfsencoding()* for more information.

버전 3.7에서 변경: Return 'utf-8' in the UTF-8 mode.

sys.**getfilesystemencodeerrors**()
Return the name of the error mode used to convert between Unicode filenames and bytes filenames. The encoding name is returned from *sys.getfilesystemencoding()*.

os.fsencode() and *os.fsdecode()* should be used to ensure that the correct encoding and errors mode are used.

버전 3.6에 추가.

sys.**get_int_max_str_digits**()
Returns the current value for the *integer string conversion length limitation*. See also *sys.set_int_max_str_digits()*.

버전 3.7.14에 추가.

sys.**getrefcount**(*object*)
Return the reference count of the *object*. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to *getrefcount()*.

sys.**getrecursionlimit**()
Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by *setrecursionlimit()*.

sys.**getsizeof**(*object*[, *default*])
Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.

If given, *default* will be returned if the object does not provide means to retrieve the size. Otherwise a *TypeError* will be raised.

getsizeof() calls the object's __sizeof__ method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

See recursive sizeof recipe for an example of using *getsizeof()* recursively to find the size of containers and all their contents.

sys.**getswitchinterval**()
Return the interpreter's "thread switch interval"; see *setswitchinterval()*.

버전 3.2에 추가.

sys.**_getframe**([*depth*])
Return a frame object from the call stack. If optional integer *depth* is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, *ValueError* is raised. The default for *depth* is zero, returning the frame at the top of the call stack.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.**getprofile**()
Get the profiler function as set by *setprofile()*.

sys.**gettrace**()
Get the trace function as set by *settrace()*.

CPython implementation detail: The *gettrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

sys.**getwindowsversion**()
Return a named tuple describing the Windows version currently running. The named elements are *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* and *platform_version*. *service_pack* contains a string, *platform_version* a 3-tuple and all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

platform will be `2 (VER_PLATFORM_WIN32_NT)`.

product_type may be one of the following values:

Constant	Meaning
1 (VER_NT_WORKSTATION)	The system is a workstation.
2 (VER_NT_DOMAIN_CONTROLLER)	The system is a domain controller.
3 (VER_NT_SERVER)	The system is a server, but not a domain controller.

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

platform_version returns the accurate major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

Availability: Windows.

버전 3.2에서 변경: Changed to a named tuple and added *service_pack_minor*, *service_pack_major*, *suite_mask*, and *product_type*.

버전 3.6에서 변경: Added *platform_version*

sys.**get_asyncgen_hooks**()
Returns an *asyncgen_hooks* object, which is similar to a *namedtuple* of the form *(firstiter, finalizer)*, where *firstiter* and *finalizer* are expected to be either `None` or functions which take an *asynchronous generator iterator* as an argument, and are used to schedule finalization of an asynchronous generator by an event loop.

버전 3.6에 추가: See **PEP 525** for more details.

참고: This function has been added on a provisional basis (see **PEP 411** for details.)

sys.**get_coroutine_origin_tracking_depth**()
Get the current coroutine origin tracking depth, as set by *set_coroutine_origin_tracking_depth()*.

버전 3.7에 추가.

참고: This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

sys.**get_coroutine_wrapper**()
Returns `None`, or a wrapper set by *set_coroutine_wrapper()*.

버전 3.5에 추가: See **PEP 492** for more details.

참고: This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

버전 3.7부터 폐지: The coroutine wrapper functionality has been deprecated, and will be removed in 3.8. See bpo-32591 for details.

sys.**hash_info**
A *named tuple* giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see 숫자 형의 해싱.

attribute	explanation
`width`	width in bits used for hash values
`modulus`	prime modulus P used for numeric hash scheme
`inf`	hash value returned for a positive infinity
`nan`	hash value returned for a nan
`imag`	multiplier used for the imaginary part of a complex number
`algorithm`	name of the algorithm for hashing of str, bytes, and memoryview
`hash_bits`	internal output size of the hash algorithm
`seed_bits`	size of the seed key of the hash algorithm

버전 3.2에 추가.

버전 3.4에서 변경: Added *algorithm*, *hash_bits* and *seed_bits*

`sys.`**`hexversion`**

The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in *`hex()`* function. The *named tuple* *`sys.version_info`* may be used for a more human-friendly encoding of the same information.

More details of `hexversion` can be found at apiabiversion.

`sys.`**`implementation`**

An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

name is the implementation's identifier, e.g. `'cpython'`. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

version is a named tuple, in the same format as *`sys.version_info`*. It represents the version of the Python *implementation*. This has a distinct meaning from the specific version of the Python *language* to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

hexversion is the implementation version in hexadecimal format, like *`sys.hexversion`*.

cache_tag is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the implementation's name and version, like `'cpython-33'`. However, a Python implementation may use some other value if appropriate. If `cache_tag` is set to `None`, it indicates that module caching should be disabled.

`sys.implementation` may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, *`sys.implementation`* will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See **PEP 421** for more information.

버전 3.3에 추가.

참고: The addition of new required attributes must go through the normal PEP process. See **PEP 421** for more information.

sys.**int_info**
A *named tuple* that holds information about Python's internal representation of integers. The attributes are read only.

Attribute	Explanation
`bits_per_digit`	number of bits held in each digit. Python integers are stored internally in base `2**int_info.bits_per_digit`
`sizeof_digit`	size in bytes of the C type used to represent a digit
`default_max_str_digits`	default value for *`sys.get_int_max_str_digits()`* when it is not otherwise explicitly configured.
`str_digits_check_threshold`	minimum non-zero value for *`sys.set_int_max_str_digits()`*, `PYTHONINTMAXSTRDIGITS`, or `-X int_max_str_digits`.

버전 3.1에 추가.

버전 3.7.14에서 변경: Added `default_max_str_digits` and `str_digits_check_threshold`.

sys.**__interactivehook__**
When this attribute exists, its value is automatically called (with no arguments) when the interpreter is launched in interactive mode. This is done after the `PYTHONSTARTUP` file is read, so that you can set this hook there. The *`site`* module *sets this*.

버전 3.4에 추가.

sys.**intern**(*string*)
Enter *string* in the table of "interned" strings and return the interned string – which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not immortal; you must keep a reference to the return value of *`intern()`* around to benefit from it.

sys.**is_finalizing**()
Return *`True`* if the Python interpreter is *shutting down*, *`False`* otherwise.

버전 3.5에 추가.

sys.**last_type**
sys.**last_value**
sys.**last_traceback**
These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger; see *`pdb`* module for more information.)

The meaning of the variables is the same as that of the return values from *`exc_info()`* above.

sys.**maxsize**
An integer giving the maximum value a variable of type `Py_ssize_t` can take. It's usually `2**31 - 1` on a 32-bit platform and `2**63 - 1` on a 64-bit platform.

sys.**maxunicode**
An integer giving the value of the largest Unicode code point, i.e. `1114111` (`0x10FFFF` in hexadecimal).

버전 3.3에서 변경: Before **PEP 393**, `sys.maxunicode` used to be either `0xFFFF` or `0x10FFFF`, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

sys.**meta_path**
A list of *meta path finder* objects that have their *find_spec()* methods called to see if one of the objects can find the module to be imported. The *find_spec()* method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's `__path__` attribute is passed in as a second argument. The method returns a *module spec*, or `None` if the module cannot be found.

더 보기:

importlib.abc.MetaPathFinder The abstract base class defining the interface of finder objects on *meta_path*.

importlib.machinery.ModuleSpec The concrete class which *find_spec()* should return instances of.

버전 3.4에서 변경: *Module specs* were introduced in Python 3.4, by **PEP 451**. Earlier versions of Python looked for a method called *find_module()*. This is still called as a fallback if a *meta_path* entry doesn't have a *find_spec()* method.

sys.**modules**
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail.

sys.**path**
A list of strings that specifies the search path for modules. Initialized from the environment variable `PYTHONPATH`, plus an installation-dependent default.

As initialized upon program startup, the first item of this list, `path[0]`, is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), `path[0]` is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted *before* the entries inserted as a result of `PYTHONPATH`.

A program is free to modify this list for its own purposes. Only strings and bytes should be added to *sys.path*; all other data types are ignored during import.

더 보기:

Module *site* This describes how to use .pth files to extend *sys.path*.

sys.**path_hooks**
A list of callables that take a path argument to try to create a *finder* for the path. If a finder can be created, it is to be returned by the callable, else raise *ImportError*.

Originally specified in **PEP 302**.

sys.**path_importer_cache**
A dictionary acting as a cache for *finder* objects. The keys are paths that have been passed to *sys.path_hooks* and the values are the finders that are found. If a path is a valid file system path but no finder is found on *sys.path_hooks* then `None` is stored.

Originally specified in **PEP 302**.

버전 3.3에서 변경: `None` is stored instead of *imp.NullImporter* when no finder is found.

`sys.`**platform**
This string contains a platform identifier that can be used to append platform-specific components to *sys.path*, for instance.

For Unix systems, except on Linux, this 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'` or `'freebsd8'`, *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('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

For other systems, the values are:

System	`platform` value
Linux	`'linux'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
Mac OS X	`'darwin'`

버전 3.3에서 변경: On Linux, *sys.platform* doesn't contain the major version anymore. It is always `'linux'`, instead of `'linux2'` or `'linux3'`. Since older Python versions include the version number, it is recommended to always use the `startswith` idiom presented above.

더 보기:

os.name has a coarser granularity. *os.uname()* gives system-dependent version information.

The *platform* module provides detailed checks for the system's identity.

`sys.`**prefix**
A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string `'/usr/local'`. This can be set at build time with the `--prefix` argument to the **configure** script. The main collection of Python library modules is installed in the directory *prefix*`/lib/python`*X.Y* while the platform independent header files (all except `pyconfig.h`) are stored in *prefix*`/include/python`*X.Y*, where *X.Y* is the version number of Python, for example `3.2`.

참고: If a *virtual environment* is in effect, this value will be changed in `site.py` to point to the virtual environment. The value for the Python installation will still be available, via *base_prefix*.

`sys.`**ps1**
`sys.`**ps2**
Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`. If a non-string object is assigned to either variable, its *str()* is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

`sys.`**setcheckinterval**(*interval*)
Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is `100`, meaning the check is performed every

100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

버전 3.2부터 폐지: This function doesn't have an effect anymore, as the internal logic for thread switching and asynchronous tasks has been rewritten. Use *setswitchinterval()* instead.

sys.**setdlopenflags**(*n*)
Set the flags used by the interpreter for `dlopen()` calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(os.RTLD_GLOBAL)`. Symbolic names for the flag values can be found in the *os* module (`RTLD_xxx` constants, e.g. *os.RTLD_LAZY*).

Availability: Unix.

sys.**set_int_max_str_digits**(*n*)
Set the *integer string conversion length limitation* used by this interpreter. See also *get_int_max_str_digits()*.

버전 3.7.14에 추가.

sys.**setprofile**(*profilefunc*)
Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter *The Python Profilers* for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see *settrace()*), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`. Error in the profile function will cause itself unset.

Profile functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'return'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *arg* depends on the event type.

The events have the following meaning:

'call' A function is called (or some other code block entered). The profile function is called; *arg* is `None`.

'return' A function (or other code block) is about to return. The profile function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised.

'c_call' A C function is about to be called. This may be an extension function or a built-in. *arg* is the C function object.

'c_return' A C function has returned. *arg* is the C function object.

'c_exception' A C function has raised an exception. *arg* is the C function object.

sys.**setrecursionlimit**(*limit*)
Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a *RecursionError* exception is raised.

버전 3.5.1에서 변경: A *RecursionError* exception is now raised if the new limit is too low at the current recursion depth.

sys.**setswitchinterval**(*interval*)
Set the interpreter's thread switch interval (in seconds). This floating-point value determines the ideal duration of

the "timeslices" allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system's decision. The interpreter doesn't have its own scheduler.

버전 3.2에 추가.

sys.**settrace**(*tracefunc*)

Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must register a trace function using *settrace()* for each thread being debugged or use *threading.settrace()*.

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.

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used for the new scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

If there is any error occurred in the trace function, it will be unset, just like `settrace(None)` is called.

The events have the following meaning:

'call' A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

'line' The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting `f_trace_lines` to *False* on that frame.

'return' A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.

'exception' An exception has occurred. The local trace function is called; *arg* is a tuple `(exception, value, traceback)`; the return value specifies the new local trace function.

'opcode' The interpreter is about to execute a new opcode (see *dis* for opcode details). The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting `f_trace_opcodes` to *True* on the frame.

Note that as an exception is propagated down the chain of callers, an `'exception'` event is generated at each level.

For more fine-grained usage, it's possible to set a trace function by assigning `frame.f_trace = tracefunc` explicitly, rather than relying on it being set indirectly via the return value from an already installed trace function. This is also required for activating the trace function on the current frame, which *settrace()* doesn't do. Note that in order for this to work, a global tracing function must have been installed with *settrace()* in order to enable the runtime tracing machinery, but it doesn't need to be the same tracing function (e.g. it could be a low overhead tracing function that simply returns `None` to disable itself immediately on each frame).

For more information on code and frame objects, refer to types.

CPython implementation detail: The *settrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

버전 3.7에서 변경: `'opcode'` event type added; `f_trace_lines` and `f_trace_opcodes` attributes added to frames

sys.**set_asyncgen_hooks**(*firstiter*, *finalizer*)
Accepts two optional keyword arguments which are callables that accept an *asynchronous generator iterator* as an argument. The *firstiter* callable will be called when an asynchronous generator is iterated for the first time. The *finalizer* will be called when an asynchronous generator is about to be garbage collected.

버전 3.6에 추가: See **PEP 525** for more details, and for a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py

참고: This function has been added on a provisional basis (see **PEP 411** for details.)

sys.**set_coroutine_origin_tracking_depth**(*depth*)
Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be None.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, pass set *depth* to zero.

This setting is thread-specific.

버전 3.7에 추가.

참고: This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

sys.**set_coroutine_wrapper**(*wrapper*)
Allows intercepting creation of *coroutine* objects (only ones that are created by an `async def` function; generators decorated with *types.coroutine()* or *asyncio.coroutine()* will not be intercepted).

The *wrapper* argument must be either:

- a callable that accepts one argument (a coroutine object);
- `None`, to reset the wrapper.

If called twice, the new wrapper replaces the previous one. The function is thread-specific.

The *wrapper* callable cannot define new coroutines directly or indirectly:

```
def wrapper(coro):
    async def wrap(coro):
        return await coro
    return wrap(coro)
sys.set_coroutine_wrapper(wrapper)

async def foo():
    pass

# The following line will fail with a RuntimeError, because
# ``wrapper`` creates a ``wrap(coro)`` coroutine:
foo()
```

See also *get_coroutine_wrapper()*.

버전 3.5에 추가: See **PEP 492** for more details.

참고: This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

버전 3.7부터 폐지: The coroutine wrapper functionality has been deprecated, and will be removed in 3.8. See bpo-32591 for details.

sys.**_enablelegacywindowsfsencoding**()
Changes the default filesystem encoding and errors mode to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the PYTHONLEGACYWINDOWSFSENCODING environment variable before launching Python.

Availability: Windows.

버전 3.6에 추가: See **PEP 529** for more details.

sys.**stdin**
sys.**stdout**
sys.**stderr**
File objects used by the interpreter for standard input, output and errors:

- stdin is used for all interactive input (including calls to *input()*);
- stdout is used for the output of *print()* and *expression* statements and for the prompts of *input()*;
- The interpreter's own prompts and its error messages go to stderr.

These streams are regular *text files* like those returned by the *open()* function. Their parameters are chosen as follows:

- The character encoding is platform-dependent. Non-Windows platforms use the locale encoding (see *locale.getpreferredencoding()*).

  On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where isatty() returns True) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system locale encoding if the process is not initially attached to a console.

  The special behaviour of the console can be overridden by setting the environment variable PYTHONLEGACYWINDOWSSTDIO before starting Python. In that case, the console codepages are used as for any other character device.

  Under all platforms, you can override the character encoding by setting the PYTHONIOENCODING environment variable before starting Python or by using the new -X utf8 command line option and PYTHONUTF8 environment variable. However, for the Windows console, this only applies when PYTHONLEGACYWINDOWSSTDIO is also set.

- When interactive, stdout and stderr streams are line-buffered. Otherwise, they are block-buffered like regular text files. You can override this value with the -u command-line option.

참고: To write or read binary data from/to the standard streams, use the underlying binary *buffer* object. For example, to write bytes to *stdout*, use sys.stdout.buffer.write(b'abc').

However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like *io.StringIO* which do not support the buffer attribute.

sys.**__stdin__**
sys.**__stdout__**
sys.**__stderr__**
These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

참고: Under some conditions `stdin`, `stdout` and `stderr` as well as the original values `__stdin__`, `__stdout__` and `__stderr__` can be `None`. It is usually the case for Windows GUI apps that aren't connected to a console and Python apps started with **pythonw**.

---

sys.**thread_info**
A *named tuple* holding information about the thread implementation.

Attribute	Explanation
`name`	Name of the thread implementation: • `'nt'`: Windows threads • `'pthread'`: POSIX threads • `'solaris'`: Solaris threads
`lock`	Name of the lock implementation: • `'semaphore'`: a lock uses a semaphore • `'mutex+cond'`: a lock uses a mutex and a condition variable • `None` if this information is unknown
version	Name and version of the thread library. It is a string, or `None` if this information is unknown.

버전 3.3에 추가.

sys.**tracebacklimit**
When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.

sys.**version**
A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use *version_info* and the functions provided by the *platform* module.

sys.**api_version**
The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

sys.**version_info**
A tuple containing the five components of the version number: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.

버전 3.1에서 변경: Added named component attributes.

sys.**warnoptions**
This is an implementation detail of the warnings framework; do not modify this value. Refer to the *warnings* module for more information on the warnings framework.

sys.**winver**
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of *version*. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by Python.

Availability: Windows.

sys.**_xoptions**
A dictionary of the various implementation-specific flags passed through the `-X` command-line option. Option names are either mapped to their values, if given explicitly, or to *True*. Example:

```
$ ./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}
```

CPython implementation detail: This is a CPython-specific way of accessing options passed through `-X`. Other implementations may export them through other means, or not at all.

버전 3.2에 추가.

Citations

30.2 sysconfig — 파이썬의 구성 정보에 접근하기

버전 3.2에 추가.

소스 코드: Lib/sysconfig.py

---

sysconfig 모듈은 설치 경로 목록과 현재 플랫폼과 관련된 구성 변수와 같은 파이썬 구성 정보에 대한 액세스를 제공합니다.

30.2.1 구성 변수

Python 배포판에는 `Makefile` 과 `pyconfig.h` 헤더 파일이 들어 있습니다. 이 파일은 파이썬 바이너리 자체와 *distutils* 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++']
```

30.2.2 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.

distutils 또는 Distutils 기반 시스템을 사용하여 설치되는 모든 새로운 구성 요소는 파일을 올바른 장소에 복사하기 위해 같은 스킴을 따릅니다.

파이썬은 현재 7가지 스킴을 지원합니다:

- *posix_prefix*: scheme for POSIX platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- *posix_home*: scheme for POSIX platforms used when a *home* option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- *posix_user*: scheme for POSIX platforms used when a component is installed through Distutils and the *user* option is used. This scheme defines paths located under the user home directory.
- *nt*: 윈도우와 같은 NT 플랫폼을 위한 스킴.
- *nt_user*: *user* 옵션이 사용될 때 NT 플랫폼용 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.
- *include*: 플랫폼마다 다르지 않은 헤더 파일용 디렉터리.
- *platinclude*: 플랫폼마다 다른 헤더 파일용 디렉터리.
- *scripts*: 스크립트 파일용 디렉터리.
- *data*: 데이터 파일용 디렉터리.

sysconfig 는 이러한 경로를 결정하는 몇 가지 함수를 제공합니다.

sysconfig.**get_scheme_names**()
현재 *sysconfig* 에서 지원되는 모든 스킴을 포함하는 튜플을 돌려줍니다.

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 `None` 을 반환합니다.

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.

30.2.3 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)
- win32 (기타 모든 것 - 구체적으로, sys.platform이 반환됩니다)

맥 OS X 는 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.

30.2.4 `sysconfig` 를 스크립트로 사용하기

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.

30.3 builtins — 내장 객체

이 모듈은 파이썬의 모든 '내장' 식별자에 대한 직접 액세스를 제공합니다. 예를 들어, builtins.open 은 내장 함수 *open()* 의 완전한 이름입니다. 설명서는 내장 함수 와 내장 상수를 참조하세요.

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:
    '''Wrapper around a file that converts output to upper-case.'''

    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 세부 사항이므로, 파이썬의 대안 구현에서는 사용되지 않을 수 있습니다.

30.4 __main__ — 최상위 스크립트 환경

'__main__' 은 최상위 코드가 실행되는 스코프의 이름입니다. 모듈의 __name__ 은 표준 입력, 스크립트 또는 대화식 프롬프트에서 읽힐 때 '__main__' 으로 설정됩니다.

모듈은 자신의 __name__ 을 검사하여 메인 스코프에서 실행 중인지를 확인할 수 있습니다. 이 때문에 임포트될 때는 실행되지 않지만, 스크립트로 실행되거나 python -m 으로 실행될 때 조건부로 동작하는 공통 관용구를 사용할 수 있습니다:

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

패키지의 경우, __main__.py 모듈을 포함 시키면 같은 효과를 얻을 수 있습니다. 모듈의 내용은 모듈이 -m 으로 실행될 때 실행됩니다.

30.5 warnings — Warning control

Source code: Lib/warnings.py

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the *warn()* function defined in this module. (C programmers use `PyErr_WarnEx()`; see exceptionhandling for details).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling *filterwarnings()* and reset to its default state by calling *resetwarnings()*.

The printing of warning messages is done by calling *showwarning()*, which may be overridden; the default implementation of this function formats the message by calling *formatwarning()*, which is also available for use by custom implementations.

더 보기:

logging.captureWarnings() allows you to handle all warnings with the standard logging infrastructure.

30.5.1 Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings.

While these are technically *built-in exceptions*, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the *Warning* class.

The following warnings category classes are currently defined:

Class	Description
Warning	This is the base class of all warning category classes. It is a subclass of *Exception*.
UserWarning	The default category for *warn()*.
DeprecationWarning	Base category for warnings about deprecated features when those warnings are intended for other Python developers (ignored by default, unless triggered by code in __main__).
SyntaxWarning	Base category for warnings about dubious syntactic features.
RuntimeWarning	Base category for warnings about dubious runtime features.
FutureWarning	Base category for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.
PendingDeprecationWarning	Base category for warnings about features that will be deprecated in the future (ignored by default).
ImportWarning	Base category for warnings triggered during the process of importing a module (ignored by default).
UnicodeWarning	Base category for warnings related to Unicode.
BytesWarning	Base category for warnings related to *bytes* and *bytearray*.
ResourceWarning	Base category for warnings related to resource usage.

버전 3.7에서 변경: Previously *DeprecationWarning* and *FutureWarning* were distinguished based on whether a feature was being removed entirely or changing its behaviour. They are now distinguished based on their intended audience and the way they're handled by the default warnings filters.

30.5.2 The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the filter determines the disposition of the match. Each entry is a tuple of the form (*action*, *message*, *category*, *module*, *lineno*), where:

- *action* is one of the following strings:

Value	Disposition
"default"	print the first occurrence of matching warnings for each location (module + line number) where the warning is issued
"error"	turn matching warnings into exceptions
"ignore"	never print matching warnings
"always"	always print matching warnings
"module"	print the first occurrence of matching warnings for each module where the warning is issued (regardless of line number)
"once"	print only the first occurrence of matching warnings, regardless of location

- *message* is a string containing a regular expression that the start of the warning message must match. The expression is compiled to always be case-insensitive.
- *category* is a class (a subclass of *Warning*) of which the warning category must be a subclass in order to match.
- *module* is a string containing a regular expression that the module name must match. The expression is compiled to be case-sensitive.
- *lineno* is an integer that the line number where the warning occurred must match, or 0 to match all line numbers.

Since the *Warning* class is derived from the built-in *Exception* class, to turn a warning into an error we simply raise `category(message)`.

If a warning is reported and doesn't match any registered filter then the "default" action is applied (hence its name).

Describing Warning Filters

The warnings filter is initialized by `-W` options passed to the Python interpreter command line and the `PYTHONWARNINGS` environment variable. The interpreter saves the arguments for all supplied entries without interpretation in `sys.warnoptions`; the *warnings* module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

Individual warnings filters are specified as a sequence of fields separated by colons:

```
action:message:category:module:line
```

The meaning of each of these fields is as described in *The Warnings Filter*. When listing multiple filters on a single line (as for `PYTHONWARNINGS`), the individual filters are separated by commas,and the filters listed later take precedence over those listed before them (as they're applied left-to-right, and the most recently applied filters take precedence over earlier ones).

Commonly used warning filters apply to either all warnings, warnings in a particular category, or warnings raised by particular modules or packages. Some examples:

```
default                       # Show all warnings (even those ignored by default)
ignore                        # Ignore all warnings
error                         # Convert all warnings to errors
error::ResourceWarning        # Treat ResourceWarning messages as errors
default::DeprecationWarning   # Show DeprecationWarning messages
ignore,default:::mymodule     # Only report warnings triggered by "mymodule"
error:::mymodule[.*]          # Convert warnings to errors in "mymodule"
                              # and any subpackages of "mymodule"
```

Default Warning Filter

By default, Python installs several warning filters, which can be overridden by the `-W` command-line option, the `PYTHONWARNINGS` environment variable and calls to *filterwarnings()*.

In regular release builds, the default warning filter has the following entries (in order of precedence):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

In debug builds, the list of default warning filters is empty.

버전 3.2에서 변경: *DeprecationWarning* is now ignored by default in addition to *PendingDeprecationWarning*.

버전 3.7에서 변경: *DeprecationWarning* is once again shown by default when triggered directly by code in `__main__`.

버전 3.7에서 변경: *BytesWarning* no longer appears in the default filter list and is instead configured via *sys.warnoptions* when `-b` is specified twice.

Overriding the default filter

Developers of applications written in Python may wish to hide *all* Python level warnings from their users by default, and only display them when running tests or otherwise working on the application. The *sys.warnoptions* attribute used to pass filter configurations to the interpreter can be used as a marker to indicate whether or not warnings should be disabled:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Developers of test runners for Python code are advised to instead ensure that *all* warnings are displayed by default for the code under test, using code like:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

Finally, developers of interactive shells that run user code in a namespace other than `__main__` are advised to ensure that *DeprecationWarning* messages are made visible by default, using code like the following (where `user_ns` is the module used to execute code entered interactively):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

30.5.3 Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning (even when warnings have been explicitly configured via the command line), then it is possible to suppress the warning using the *catch_warnings* context manager:

```
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. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

30.5.4 Testing Warnings

To test warnings raised by code, use the *catch_warnings* context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
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)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once`/`default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

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. The *showwarning()* function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

30.5.5 Updating Code For New Versions of Dependencies

Warning categories that are primarily of interest to Python developers (rather than end users of applications written in Python) are ignored by default.

Notably, this "ignored by default" list includes *DeprecationWarning* (for every module except `__main__`), which means developers should make sure to test their code with typically ignored warnings made visible in order to receive timely notifications of future breaking API changes (whether in the standard library or third party packages).

In the ideal case, the code will have a suitable test suite, and the test runner will take care of implicitly enabling all warnings when running tests (the test runner provided by the *unittest* module does this).

In less ideal cases, applications can be checked for use of deprecated interfaces by passing `-Wd` to the Python interpreter (this is shorthand for `-W default`) or setting `PYTHONWARNINGS=default` in the environment. This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you can change what argument is passed to `-W` (e.g. `-W error`). See the `-W` flag for more details on what is possible.

30.5.6 Available Functions

warnings.**warn**(*message*, *category=None*, *stacklevel=1*, *source=None*)

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. Alternatively *message* can be a `Warning` instance, in which case *category* will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

버전 3.6에서 변경: Added *source* parameter.

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. *message* must be a string and *category* a subclass of `Warning` or *message* may be a `Warning` instance, in which case *category* will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

버전 3.6에서 변경: Add the *source* parameter.

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to *file*, which defaults to `sys.stderr`. You may replace this function with any callable by assigning to `warnings.showwarning`. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `showwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*, *line=None*)

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `formatwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.**filterwarnings**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

Insert an entry into the list of *warnings filter specifications*. The entry is inserted at the front by default; if *append* is true, it is inserted at the end. This checks the types of the arguments, compiles the *message* and *module* regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

warnings.**simplefilter**(*action*, *category=Warning*, *lineno=0*, *append=False*)

Insert a simple entry into the list of *warnings filter specifications*. The meaning of the function parameters is as for `filterwarnings()`, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

warnings.**resetwarnings**()
Reset the warnings filter. This discards the effect of all previous calls to *filterwarnings()*, including that of the -W command line options and calls to *simplefilter()*.

30.5.7 Available Context Managers

class warnings.**catch_warnings**(*, *record=False*, *module=None*)
A context manager that copies and, upon exit, restores the warnings filter and the *showwarning()* function. If the *record* argument is *False* (the default) the context manager returns *None* on entry. If *record* is *True*, a list is returned that is progressively populated with objects as seen by a custom *showwarning()* function (which also suppresses output to sys.stdout). Each object in the list has attributes with the same names as the arguments to *showwarning()*.

The *module* argument takes a module that will be used instead of the module returned when you import *warnings* whose filter will be protected. This argument exists primarily for testing the *warnings* module itself.

참고: The *catch_warnings* manager works by replacing and then later restoring the module's *showwarning()* function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.

30.6 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:
    '''Class for keeping track of an item in inventory.'''
    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.

버전 3.7에 추가.

30.6.1 모듈 수준의 데코레이터, 클래스 및 함수

@dataclasses.**dataclass**(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*)

이 함수는 (아래에서 설명하는) 생성된 특수 메서드를 클래스에 추가하는데 사용되는 데코레이터 입니다.

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 "따분한" 메서드들을 클래스에 추가하는데, 아래에서 설명합니다. 추가할 메서드가 클래스에 이미 존재하면, 동작은 매개 변수에 따라 다른데, 아래에 문서화됩니다. 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)
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. repr에서 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`: `False` (기본값) 면 : `eq` 와 `frozen` 의 설정에 따라 `__hash__()` 메서드가 생성됩니다.

  `__hash__()` 는 내장 *hash()* 에 의해 사용되며, 딕셔너리와 집합 같은 해시 컬렉션에 객체가 추가될 때 사용됩니다. `__hash__()` 를 갖는다는 것은 클래스의 인스턴스가 불변이라는 것을 의미합

니다. 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, 그런데도 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, 슈퍼클래스의 __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. If __setattr__() or __delattr__() is defined in the class, then *TypeError* is raised. See the discussion below.

필드는 선택적으로 일반적인 파이썬 문법을 사용하여 기본값을 지정할 수 있습니다:

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

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 클래스에서 일어날 수도 있고, 클래스 상속의 결과일 때도 마찬가지입니다.

dataclasses.**field**(*, *default=MISSING*, *default_factory=MISSING*, *repr=True*, *hash=None*, *init=True*, *compare=True*, *metadata=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 default 와 default_factory 매개변수가 제공되는지를 탐지하는데 사용되는 표지 객체입니다. None 이 default 에 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.
- compare: 참(기본값)이면, 이 필드는 생성된 같음 및 비교 메서드(__eq__(), __gt__() 등)에 포함됩니다.
- hash: 이것은 bool 또는 None 일 수 있습니다. 참이면, 이 필드는 생성된 __hash__() 메서드에 포함됩니다. None (기본값) 이면, compare 의 값을 사용합니다. 이것은 일반적으로 기대되는 행동입니다. 필드가 비교에 사용되면 해시에서 고려해야 합니다. 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.
- metadata: 매핑이나 None이 될 수 있습니다. None은 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.

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. 문서화된 어트리뷰트는 다음과 같습니다:

- name: 필드의 이름.
- type: 필드의 형.
- default, default_factory, init, repr, hash, compare, metadata 는 *field()* 선언에서와 같은 의미와 값을 가지고 있습니다.

다른 어트리뷰트도 있을 수 있지만, 내부적인 것이므로 검사하거나 의존해서는 안 됩니다.

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**(*instance*, *, *dict_factory=dict*)

데이터 클래스 instance 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. 예를 들면:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: List[Point]

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}]}
```

instance 가 데이터 클래스 인스턴스가 아닌 경우 *TypeError* 를 일으킵니다.

dataclasses.**astuple**(*instance*, *, *tuple_factory=tuple*)

데이터 클래스 instance 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.

이전 예에서 계속하면:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

instance 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*)

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 의 값은 *dataclass()* 에서와 같은 의미가 있습니다.

이 함수가 꼭 필요하지는 않습니다. 임의의 파이썬 메커니즘으로 __annotations__ 을 갖는 새 클래스를 만든 후에 *dataclass()* 함수를 적용하면 데이터 클래스로 변환되기 때문입니다. 이 함수는 편의상 제공됩니다. 예를 들어:

```
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
    y: 'typing.Any'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    z: int = 5

    def add_one(self):
        return self.x + 1
```

dataclasses.**replace**(*instance*, ***changes*)

instance 와 같은 형의 새 객체를 만드는데, 필드를 changes 의 값들로 대체합니다. instance 가 데이터 클래스가 아니라면 *TypeError* 를 발생시킵니다. changes 의 값이 필드를 지정하지 않으면 *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.

dataclasses.**is_dataclass**(*class_or_instance*)

Return True if its parameter is a dataclass or an instance of one, otherwise return False.

(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)
```

30.6.2 초기화 후처리

클래스에 __post_init__() 가 정의된 경우, 생성된 __init__() 코드는 __post_init__() 메서드를 호출합니다. 일반적으로 self.__post_init__() 로 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
```

매개변수를 __post_init__() 에 전달하는 방법은 초기화 전용 변수에 대한 아래 섹션을 참조하십시오. 또한 *replace()* 가 init=False 필드를 처리하는 방식에 관한 경고를 보십시오.

30.6.3 클래스 변수

dataclass() 가 실제로 필드의 형을 검사하는 두 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.

30.6.4 초기화 전용 변수

dataclass() 가 형 어노테이션을 검사하는 다른 한 곳은 필드가 초기화 전용 변수인지 확인하는 것입니다. 필드의 형이 `dataclasses.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
    database: InitVar[DatabaseType] = 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.

30.6.5 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.

30.6.6 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
@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):
```

30.6.7 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.

30.6.8 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 = []
    def add(self, element):
        self.x += element
```

비슷한 코드를 생성합니다:

```
class D:
    x = []
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x += 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 같은 x 를 공유합니다. 데이터 클래스는 일반적인 파이썬 클래스 생성을 사용하기 때문에, 이 동작 역시 공유합니다. 데이터 클래스가 이 조건을 감지하는 일반적인 방법은 없습니다. 대신, 데이터 클래스는 list, dict, set 형의 기본 매개변수를 탐지하면 *TypeError* 를 발생시킵니다. 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
```

30.6.9 예외

exception dataclasses.**FrozenInstanceError**

frozen=True 로 정의된 데이터 클래스에서 묵시적으로 정의된 __setattr__() 또는 __delattr__() 이 호출될 때 발생합니다.

30.7 contextlib — Utilities for with-statement contexts

Source code: Lib/contextlib.py

This module provides utilities for common tasks involving the with statement. For more information see also 컨텍스트 관리자 형 and context-managers.

30.7.1 Utilities

Functions and classes provided:

class contextlib.**AbstractContextManager**

An *abstract base class* for classes that implement object.__enter__() and object.__exit__(). A default implementation for object.__enter__() is provided which returns self while object.__exit__() is an abstract method which by default returns None. See also the definition of 컨텍스트 관리자 형.

버전 3.6에 추가.

class contextlib.**AbstractAsyncContextManager**

An *abstract base class* for classes that implement object.__aenter__() and object.__aexit__().

A default implementation for `object.__aenter__()` is provided which returns `self` while `object.__aexit__()` is an abstract method which by default returns `None`. See also the definition of async-context-managers.

버전 3.7에 추가.

`@contextlib.`**`contextmanager`**

This function is a *decorator* that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a `close()` method for use with `contextlib.closing`

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a *generator*-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try…except…finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

`contextmanager()` uses *`ContextDecorator`* so the context managers it creates can be used as decorators as well as in `with` statements. When used as a decorator, a new generator instance is implicitly created on each function call (this allows the otherwise "one-shot" context managers created by *`contextmanager()`* to meet the requirement that context managers support multiple invocations in order to be used as decorators).

버전 3.2에서 변경: Use of *`ContextDecorator`*.

`@contextlib.`**`asynccontextmanager`**

Similar to *`contextmanager()`*, but creates an asynchronous context manager.

This function is a *decorator* that can be used to define a factory function for `async with` statement asynchronous context managers, without needing to create a class or separate `__aenter__()` and `__aexit__()` methods. It must be applied to an *asynchronous generator* function.

A simple example:

```
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 ...')
```

버전 3.7에 추가.

contextlib.**closing**(*thing*)

Return a context manager that closes *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

contextlib.**nullcontext**(*enter_result=None*)

Return a context manager that returns *enter_result* from `__enter__`, but otherwise does nothing. It is intended to be used as a stand-in for an optional context manager, for example:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

An example using *enter_result*:

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

버전 3.7에 추가.

contextlib.**suppress**(**exceptions*)

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a with statement and then resumes execution with the first statement following the end of the with statement.

As with any other mechanism that completely suppresses exceptions, this context manager should be used only to cover very specific errors where silently continuing with program execution is known to be the right thing to do.

For example:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

This code is equivalent to:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

This context manager is *reentrant*.

버전 3.4에 추가.

contextlib.**redirect_stdout**(*new_target*)

Context manager for temporarily redirecting *sys.stdout* to another file or file-like object.

This tool adds flexibility to existing functions or classes whose output is hardwired to stdout.

For example, the output of *help()* normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an *io.StringIO* object:

```
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

To send the output of *help()* to a file on disk, redirect the output to a regular file:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

To send the output of `help()` to *sys.stderr*:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Note that the global side effect on `sys.stdout` means that this context manager is not suitable for use in library code and most threaded applications. It also has no effect on the output of subprocesses. However, it is still a useful approach for many utility scripts.

This context manager is *reentrant*.

버전 3.4에 추가.

contextlib.**redirect_stderr**(*new_target*)

Similar to `redirect_stdout()` but redirecting `sys.stderr` to another file or file-like object.

This context manager is *reentrant*.

버전 3.5에 추가.

class contextlib.**ContextDecorator**

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by `contextmanager()`, so you get this functionality automatically.

Example of `ContextDecorator`:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @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
```

This change is just syntactic sugar for any construct of the following form:

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` lets you instead write:

```
@cm()
def f():
    # Do stuff
```

It makes it clear that the `cm` applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using `ContextDecorator` as a mixin class:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

참고: As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit `with` statement inside the function should be used.

버전 3.2에 추가.

class contextlib.**ExitStack**

A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single with statement as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

Each instance maintains a stack of registered callbacks that are called in reverse order when the instance is closed (either explicitly or implicitly at the end of a `with` statement). Note that callbacks are *not* invoked implicitly when the context stack instance is garbage collected.

This stack model is used so that context managers that acquire their resources in their `__init__` method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested `with` statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways.

버전 3.3에 추가.

enter_context(*cm*)
Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.

push(*exit*)
Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

callback(*callback, *args, **kwds*)
Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

pop_all()
Transfers the callback stack to a fresh *ExitStack* instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an "all or nothing" operation as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()
Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

class contextlib.**AsyncExitStack**
An asynchronous context manager, similar to *ExitStack*, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

The `close()` method is not implemented, *aclose()* must be used instead.

enter_async_context(*cm*)
Similar to `enter_context()` but expects an asynchronous context manager.

push_async_exit(*exit*)
Similar to `push()` but expects either an asynchronous context manager or a coroutine function.

push_async_callback(*callback*, **args*, ***kwds*)
Similar to `callback()` but expects a coroutine function.

aclose()
Similar to `close()` but properly handles awaitables.

Continuing the example for *asynccontextmanager()*:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

버전 3.7에 추가.

30.7.2 Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by *contextlib*.

Supporting a variable number of context managers

The primary use case for *ExitStack* is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single `with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```
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)
    # Perform operations that use the acquired resources
```

As shown, *ExitStack* also makes it quite easy to use `with` statements to manage arbitrary resources that don't natively support the context management protocol.

Catching exceptions from __enter__ methods

It is occasionally desirable to catch exceptions from an `__enter__` method implementation, *without* inadvertently catching exceptions from the `with` statement body or the context manager's `__exit__` method. By using *ExitStack* the steps in the context management protocol can be separated slightly in order to allow this:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with `try/except/finally` statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then *ExitStack* can make it easier to handle various situations that can't be handled directly in a `with` statement.

Cleaning up in an `__enter__` implementation

As noted in the documentation of *ExitStack.push()*, this method can be useful in cleaning up an already allocated resource if later steps in the `__enter__()` implementation fail.

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
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
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            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):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

ExitStack makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup up behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, *args, **kwds):
        super(Callback, self).__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()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of *ExitStack.callback()* to declare the resource cleanup in advance:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

ContextDecorator makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from *ContextDecorator* provides both capabilities in a single definition:

```
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)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

더 보기:

PEP 343 - The "with" statement The specification, background, and examples for the Python `with` statement.

30.7.3 Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they're used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using *contextmanager()* are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be "reentrant". These context managers can not only be used in multiple `with` statements, but may also be used *inside* a `with` statement that is already using the same context manager.

threading.RLock is an example of a reentrant context manager, as are *suppress()* and *redirect_stdout()*. Here's a very simple example of reentrant use:

```
>>> 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
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. *redirect_stdout()*, for example, is definitely not thread safe, as it makes a global modification to the system state by binding *sys.stdout* to a different stream.

Reusable context managers

Distinct from both single use and reentrant context managers are "reusable" context managers (or, to be completely explicit, "reusable, but not reentrant" context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

threading.Lock is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use *threading.RLock* instead).

Another example of a reusable, but not reentrant, context manager is *ExitStack*, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```
>>> 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
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate *ExitStack* instances instead of reusing a single instance avoids that problem:

```
>>> 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")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

30.8 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 *ABC*에서 파생시켜서 만들 수 있습니다. 예를 들어:

```
from abc import ABC

class MyABC(ABC):
    pass
```

ABC 의 형은 여전히 *ABCMeta* 이므로, *ABC*를 상속할 때는 메타 클래스 사용에 관한 일반적인 주의가 필요한데, 다중 상속이 메타 클래스 충돌을 일으킬 수 있기 때문입니다. metaclass 키워드를 전달하고 *ABCMeta*를 직접 사용해서 추상 베이스 클래스를 정의할 수도 있습니다, 예를 들어:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

버전 3.4에 추가.

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 믹스 인 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]

[1] C++ 프로그래머는 파이썬의 가상 베이스 클래스 개념이 C++과 다르다는 것을 알아야 합니다.

*ABCMeta*를 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 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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 호출할 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 지원되지 않습니다. *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, ...):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    @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**

버전 3.2에 추가.

버전 3.3부터 폐지: 이제 *classmethod*와 *abstractmethod()*를 함께 사용할 수 있어서, 이 데코레이터는 필요 없습니다.

내장 *classmethod()*의 서브 클래스로, 추상 classmethod를 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, ...):
        ...
```

@abc.**abstractstaticmethod**

버전 3.2에 추가.

버전 3.3부터 폐지: 이제 *staticmethod*와 *abstractmethod()*를 함께 사용할 수 있어서, 이 데코레이터는 필요 없습니다.

내장 *staticmethod()*의 서브 클래스로, 추상 staticmethod를 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(...):
        ...
```

@abc.**abstractproperty**

버전 3.3부터 폐지: 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
    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.

버전 3.4에 추가.

30.9 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.

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.

atexit.**unregister**(*func*)

인터프리터 종료 시 실행할 함수 목록에서 *func* 를 제거합니다. *unregister()*를 호출한 후에는, 인터프리터가 종료할 때 *func* 이 호출되지 않음이 보장됩니다. 두 번 이상 등록했더라도 그렇습니다. *unregister()* 는 *func* 가 이전에 등록되지 않았다면 아무것도 하지 않습니다.

더 보기:

모듈 *readline* *readline* 히스토리 파일을 읽고 쓰는 *atexit* 의 유용한 예.

30.9.1 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
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.

30.10 traceback — 스택 트레이스백 인쇄와 조회

소스 코드: Lib/traceback.py

이 모듈은 파이썬 프로그램의 스택 트레이스를 추출, 포맷 및 인쇄하는 표준 인터페이스를 제공합니다. 스택 트레이스를 인쇄할 때 파이썬 인터프리터의 동작을 정확하게 모방합니다. 이것은 가령 인터프리터를 둘러싸는 "래퍼"처럼 프로그램 제어 하에서 스택 트레이스를 인쇄하려고 할 때 유용합니다.

이 모듈은 트레이스백 객체를 사용합니다 — 이는 *sys.last_traceback* 변수에 저장되고 *sys.exc_info()*의 세 번째 항목으로 반환되는 객체 형입니다.

이 모듈은 다음 함수를 정의합니다:

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.

버전 3.5에서 변경: 음수 *limit* 지원을 추가했습니다.

traceback.**print_exception**(*etype*, *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 *etype*과 *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.

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.

traceback.**print_exc**(*limit=None*, *file=None*, *chain=True*)
이것은 `print_exception(*sys.exc_info(), limit, file, chain)`의 줄임 표현입니다.

traceback.**print_last**(*limit=None*, *file=None*, *chain=True*)
이 것 은 `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)`의 줄임 표현입니다. 일반적으로 예외가 대화식 프롬프트에 도달한 후에만 작동합니다(*sys.last_type*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. `line`은 선행과 후행 공백이 제거된 문자열입니다; 소스를 사용할 수 없으면 `None`입니다.

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.**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**(*etype*, *value*)
트레이스백의 예외 부분을 포맷합니다. 인자는 `sys.last_type`과 `sys.last_value`에서 제공하는 것과 같은 예외 형과 값입니다. 반환 값은 각각 줄 바꿈으로 끝나는 문자열의 리스트입니다. 일반적으로, 리스트는 단일 문자열을 포함합니다; 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.

traceback.**format_exception**(*etype*, *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.

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.

버전 3.4에 추가.

traceback.**walk_stack**(*f*)
주어진 프레임에서 f.f_back을 따라 스택을 걸어가며 각 프레임의 프레임과 줄 번호를 산출(yield)합니다. *f*가 None이면, 현재 스택이 사용됩니다. 이 도우미는 *StackSummary.extract()*와 함께 사용됩니다.

버전 3.5에 추가.

traceback.**walk_tb**(*tb*)
tb_next를 따라 트레이스백을 걸으면서 각 프레임의 프레임과 줄 번호를 산출(yield)합니다. 이 도우미는 *StackSummary.extract()*와 함께 사용됩니다.

버전 3.5에 추가.

이 모듈은 또한 다음과 같은 클래스를 정의합니다:

30.10.1 TracebackException 객체

버전 3.5에 추가.

TracebackException 객체는 실제 예외에서 만들어져 나중에 인쇄하기 위한 데이터를 경량 방식으로 포착합니다.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
나중에 렌더링하기 위해 예외를 포착합니다. *limit*, *lookup_lines* 및 *capture_locals*는 *StackSummary* 클래스와 같습니다.

locals가 포착되면, 트레이스백에도 표시됨에 유의하십시오.

__cause__
원래 __cause__의 *TracebackException*.

__context__
원래 __context__의 *TracebackException*.

__suppress_context__
원래 예외의 __suppress_context__ 값.

stack
트레이스백을 나타내는 *StackSummary*.

exc_type
원래 트레이스백의 클래스.

filename
문법 에러일 때 - 에러가 발생한 파일 이름.

lineno
문법 에러일 때 - 에러가 발생한 줄 번호.

text
문법 에러일 때 - 에러가 발생한 텍스트.

offset
문법 에러일 때 - 에러가 발생한 텍스트에서의 오프셋.

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.

locals가 포착되면, 트레이스백에도 표시됨에 유의하십시오.

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.

어떤 예외가 발생했는지를 나타내는 메시지는 항상 출력의 마지막 문자열입니다.

format_exception_only()
트레이스백의 예외 부분을 포맷합니다.

반환 값은 각각 줄 바꿈으로 끝나는 문자열의 제너레이터입니다.

일반적으로, 제너레이터는 단일 문자열을 방출합니다; 그러나 *SyntaxError* 예외의 경우, 문법 에러가 발생한 위치에 대한 자세한 정보를 (인쇄할 때) 표시하는 여러 줄을 방출합니다.

어떤 예외가 발생했는지를 나타내는 메시지는 항상 출력의 마지막 문자열입니다.

30.10.2 StackSummary 객체

버전 3.5에 추가.

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.

classmethod from_list(*a_list*)
제공된 *FrameSummary* 객체의 리스트나 이전 스타일의 튜플 리스트로 *StackSummary* 객체를 생성합니다. 각 튜플은 파일명(filename), 줄 번호(lineno), 이름(name), 줄(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.

30.10.3 FrameSummary 객체

버전 3.5에 추가.

FrameSummary 객체는 트레이스백에서 단일 프레임을 나타냅니다.

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)
포맷되거나 인쇄 중인 트레이스백이나 스택의 단일 프레임을 나타냅니다. 선택적으로 문자열로 변환된 버전의 프레임 지역 변수를 포함할 수 있습니다. *lookup_line*이 False이면, *FrameSummary*의 line 어트리뷰트에 액세스할 때까지 (튜플로 캐스트 할 때도 발생합니다) 소스 코드를 찾지 않습니다. line은 직접 제공될 수 있으며, 줄 조회가 전혀 발생하지 않도록 합니다. *locals*는 선택적 지역 변수 딕셔너리이며, 제공되면 변수 표현(representation)은 나중에 표시할 수 있도록 요약에 저장됩니다.

30.10.4 트레이스백 예제

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_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    # exc_type below is ignored on 3.5 and later
    print(repr(traceback.format_exception(exc_type, exc_value,
                                          exc_traceback)))
    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_death()
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_death()
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...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\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_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

다음 예제는 스택을 인쇄하고 포맷하는 다양한 방법을 보여줍니다:

```
>>> import traceback
>>> def another_function():
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     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(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

30.11 __future__ — 퓨처 문 정의

소스 코드: Lib/__future__.py

__future__ 는 실제 모듈이며 세 가지 용도로 사용됩니다:

- 임포트 문을 분석하고 임포트하는 모듈을 발견하리라고 기대하는 기존 도구가 혼동하지 않게 하려고.
- 2.1 이전의 배포에서 퓨처 문 을 실행하면 최소한 실행시간 예외를 일으키도록 보장하기 위해 (2.1 이전에는 그런 이름의 모듈이 없으므로 *__future__* 임포트는 실패합니다).
- 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.

__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
)
```

OptionalRelease 는 해당 기능이 승인된 첫 번째 배포를 기록합니다.

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 삭제되었음을 의미합니다.

클래스 `_Feature` 의 인스턴스는 두 개의 상응하는 메서드인 `getOptionalRelease()` 와 `getMandatoryRelease()` 를 가지고 있습니다.

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 `compiler_flag` 어트리뷰트에 저장됩니다.

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
nested_scopes	2.1.0b1	2.2	**PEP 227**: 정적으로 중첩된 스코프
generators	2.2.0a1	2.3	**PEP 255**: 단순 제너레이터
division	2.2.0a2	3.0	**PEP 238**: 나누기 연산자 변경
absolute_import	2.5.0a1	3.0	**PEP 328**: 임포트: 복수 줄 및 절대/상대
with_statement	2.5.0a1	2.6	**PEP 343**: "*with*" 문
print_function	2.6.0a2	3.0	**PEP 3105**: *print*를 함수로 만들기
unicode_literals	2.6.0a2	3.0	**PEP 3112**: 파이썬 *3000*의 바이트열 리터럴
generator_stop	3.5.0b1	3.7	**PEP 479**: 제너레이터 내부의 *StopIteration* 처리
annotations	3.7.0b1	4.0	**PEP 563**: 어노테이션의 지연된 평가

더 보기:

future 컴파일러가 퓨처 임포트를 처리하는 방법.

30.12 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 디버그하려면, `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`**`()`
Return `True` if automatic collection is enabled.

`gc.`**`collect`**`(`*generation=2*`)`
인자가 없으면, 전체 수거를 실행합니다. 선택적 인자 *generation*은 어떤 세대를 수거할지 지정하는 정수(0에서 2)일 수 있습니다. 세대 번호가 유효하지 않으면 *ValueError*가 발생합니다. 발견된 도달할 수 없는(unreachable) 객체의 수가 반환됩니다.

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.

`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`**`()`
Returns a list of all objects tracked by the collector, excluding the list returned.

`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.

버전 3.4에 추가.

`gc.`**`set_threshold`**`(`*threshold0*`[`*, threshold1*`[`*, threshold2*`]]`)
가비지 수거 임곗값(수거 빈도)을 설정합니다. *threshold0*을 0으로 설정하면 수거가 비활성화됩니다.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation `0`). If an object survives a collection it is moved into the next older generation. Since generation `2` is the oldest generation, objects in that generation remain there

after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. Initially only generation `0` is examined. If generation `0` has been examined more than *threshold1* times since generation `1` has been examined, then generation `1` is examined as well. With the third generation, things are a bit more complicated, see Collecting the oldest generation for more information.

`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.

`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.

`gc.`**`is_tracked`**`(`*obj*`)`
가비지 수거기가 객체를 현재 추적하고 있으면 `True`를, 그렇지 않으면 `False`를 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})
False
>>> gc.is_tracked({"a": []})
True
```

버전 3.1에 추가.

`gc.`**`freeze`**`()`
Freeze all the objects tracked by gc - move them to a permanent generation and ignore all the future collections. This can be used before a POSIX fork() call to make the gc copy-on-write friendly or to speed up collection. Also collection before a POSIX fork() call may free pages for future allocation which can cause copy-on-write too so it's advised to disable gc in master process and freeze before fork and enable gc in child process.

버전 3.7에 추가.

gc.**unfreeze**()
영구 세대(permanent generation)의 객체를 고정 해제하고, 가장 나이 든 세대로 되돌립니다.

버전 3.7에 추가.

gc.**get_freeze_count**()
영구 세대(permanent generation)에 있는 객체 수를 반환합니다.

버전 3.7에 추가.

다음 변수가 전용 액세스로 제공됩니다(값을 변경할 수는 있지만, 다시 연결해서는 안 됩니다):

gc.**garbage**
A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). Starting with Python 3.4, this list should be empty most of the time, except when using instances of C extension types with a non-NULL `tp_del` slot.

*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.

버전 3.3에 추가.

*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_SAVEALL과 같습니다).

30.13 inspect — Inspect live objects

Source code: Lib/inspect.py

The *inspect* module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

30.13.1 Types and members

The *getmembers()* function retrieves the members of an object such as a class or module. The functions whose names begin with "is" are mainly provided as convenient choices for the second argument to *getmembers()*. They also help you determine when you can expect to find the following special attributes:

Type	Attribute	Description
module	__doc__	documentation string
	__file__	filename (missing for built-in modules)
class	__doc__	documentation string
	__name__	name with which this class was defined
	__qualname__	qualified name
	__module__	name of module in which this class was defined
method	__doc__	documentation string
	__name__	name with which this method was defined
	__qualname__	qualified name
	__func__	function object containing implementation of method
	__self__	instance to which this method is bound, or None
	__module__	name of module in which this method was defined
function	__doc__	documentation string
	__name__	name with which this function was defined
	__qualname__	qualified name
	__code__	code object containing compiled function *bytecode*
	__defaults__	tuple of any default values for positional or keyword parameters
	__kwdefaults__	mapping of any default values for keyword-only parameters
	__globals__	global namespace in which this function was defined
	__annotations__	mapping of parameters names to annotations; "return" key is reserved for return annotations.

다음 페이지에

표 1 – 이전 페이지에서 계속

Type	Attribute	Description
	__module__	name of module in which this function was defined
traceback	tb_frame	frame object at this level
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
frame	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_trace	tracing function for this frame, or `None`
code	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	string of raw compiled bytecode
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of `CO_*` flags, read more *here*
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_freevars	tuple of names of free variables (referenced via a function's closure)
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	name with which this code object was defined
	co_names	tuple of names of local variables
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
generator	__name__	name
	__qualname__	qualified name
	gi_frame	frame
	gi_running	is the generator running?
	gi_code	code
	gi_yieldfrom	object being iterated by `yield from`, or `None`
coroutine	__name__	name
	__qualname__	qualified name
	cr_await	object being awaited on, or `None`
	cr_frame	frame
	cr_running	is the coroutine running?
	cr_code	code
	cr_origin	where coroutine was created, or `None`. See *`sys.set_coroutine_origin_tracking_dep`*
builtin	__doc__	documentation string
	__name__	original name of this function or method
	__qualname__	qualified name
	__self__	instance to which a method is bound, or `None`

버전 3.5에서 변경: Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

버전 3.7에서 변경: Add `cr_origin` attribute to coroutines.

`inspect.`**`getmembers`**(*object*[, *predicate*])
Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional *predicate* argument is supplied, only members for which the predicate returns a true value are included.

참고: *`getmembers()`* will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

`inspect.`**`getmodulename`**(*path*)
Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in *`importlib.machinery.all_suffixes()`*. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

버전 3.3에서 변경: The function is based directly on *`importlib`*.

`inspect.`**`ismodule`**(*object*)
Return `True` if the object is a module.

`inspect.`**`isclass`**(*object*)
Return `True` if the object is a class, whether built-in or created in Python code.

`inspect.`**`ismethod`**(*object*)
Return `True` if the object is a bound method written in Python.

`inspect.`**`isfunction`**(*object*)
Return `True` if the object is a Python function, which includes functions created by a *lambda* expression.

`inspect.`**`isgeneratorfunction`**(*object*)
Return `True` if the object is a Python generator function.

`inspect.`**`isgenerator`**(*object*)
Return `True` if the object is a generator.

`inspect.`**`iscoroutinefunction`**(*object*)
Return `True` if the object is a *coroutine function* (a function defined with an `async def` syntax).

버전 3.5에 추가.

`inspect.`**`iscoroutine`**(*object*)
Return `True` if the object is a *coroutine* created by an `async def` function.

버전 3.5에 추가.

`inspect.`**`isawaitable`**(*object*)
Return `True` if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

버전 3.5에 추가.

inspect.**isasyncgenfunction**(*object*)
Return True if the object is an *asynchronous generator* function, for example:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

버전 3.6에 추가.

inspect.**isasyncgen**(*object*)
Return True if the object is an *asynchronous generator iterator* created by an *asynchronous generator* function.

버전 3.6에 추가.

inspect.**istraceback**(*object*)
Return True if the object is a traceback.

inspect.**isframe**(*object*)
Return True if the object is a frame.

inspect.**iscode**(*object*)
Return True if the object is a code.

inspect.**isbuiltin**(*object*)
Return True if the object is a built-in function or a bound built-in method.

inspect.**isroutine**(*object*)
Return True if the object is a user-defined or built-in function or method.

inspect.**isabstract**(*object*)
Return True if the object is an abstract base class.

inspect.**ismethoddescriptor**(*object*)
Return True if the object is a method descriptor, but not if *ismethod()*, *isclass()*, *isfunction()* or *isbuiltin()* are true.

This, for example, is true of int.__add__. An object passing this test has a __get__() method but not a __set__() method, but beyond that the set of attributes varies. A *__name__* attribute is usually sensible, and __doc__ often is.

Methods implemented via descriptors that also pass one of the other tests return False from the *ismethoddescriptor()* test, simply because the other tests promise more – you can, e.g., count on having the __func__ attribute (etc) when an object passes *ismethod()*.

inspect.**isdatadescriptor**(*object*)
Return True if the object is a data descriptor.

Data descriptors have both a __get__ and a __set__ method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have *__name__* and __doc__ attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

inspect.**isgetsetdescriptor**(*object*)
Return True if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via PyGetSetDef structures. For Python implementations without such types, this method will always return False.

inspect.**ismemberdescriptor**(*object*)
Return True if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via PyMemberDef structures. For Python implementations without such types, this method will always return False.

30.13.2 Retrieving source code

inspect.**getdoc**(*object*)
Get the documentation string for an object, cleaned up with *cleandoc()*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy.

버전 3.5에서 변경: Documentation strings are now inherited if not overridden.

inspect.**getcomments**(*object*)
Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return None. This could happen if the object has been defined in C or the interactive shell.

inspect.**getfile**(*object*)
Return the name of the (text or binary) file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

inspect.**getmodule**(*object*)
Try to guess which module an object was defined in.

inspect.**getsourcefile**(*object*)
Return the name of the Python source file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

inspect.**getsourcelines**(*object*)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An *OSError* is raised if the source code cannot be retrieved.

버전 3.3에서 변경: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**getsource**(*object*)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An *OSError* is raised if the source code cannot be retrieved.

버전 3.3에서 변경: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**cleandoc**(*doc*)
Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

30.13.3 Introspecting callables with the Signature object

버전 3.3에 추가.

The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the *signature()* function.

inspect.**signature**(*callable*, *, *follow_wrapped=True*)

Return a *Signature* object for the given `callable`:

```
>>> 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'>
```

Accepts a wide range of Python callables, from plain functions and classes to *functools.partial()* objects.

Raises *ValueError* if no signature can be provided, and *TypeError* if that type of object is not supported.

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 3.5에 추가: `follow_wrapped` parameter. Pass `False` to get a signature of `callable` specifically (`callable.__wrapped__` will not be used to unwrap decorated callables.)

참고: Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

---

class inspect.**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)

A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a *Parameter* object in its *parameters* collection.

The optional *parameters* argument is a sequence of *Parameter* objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument, can be an arbitrary Python object, is the "return" annotation of the callable.

Signature objects are *immutable*. Use *Signature.replace()* to make a modified copy.

버전 3.5에서 변경: Signature objects are picklable and hashable.

empty

A special class-level marker to specify absence of a return annotation.

parameters

An ordered mapping of parameters' names to the corresponding *Parameter* objects. Parameters appear in strict definition order, including keyword-only parameters.

버전 3.7에서 변경: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

return_annotation
The "return" annotation for the callable. If the callable has no "return" annotation, this attribute is set to *Signature.empty*.

bind(**args*, ***kwargs*)
Create a mapping from positional and keyword arguments to parameters. Returns *BoundArguments* if `*args` and `**kwargs` match the signature, or raises a *TypeError*.

bind_partial(**args*, ***kwargs*)
Works the same way as *Signature.bind()*, but allows the omission of some required arguments (mimics *functools.partial()* behavior.) Returns *BoundArguments*, or raises a *TypeError* if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)
Create a new Signature instance based on the instance replace was invoked on. It is possible to pass different `parameters` and/or `return_annotation` to override the corresponding properties of the base signature. To remove return_annotation from the copied Signature, pass in *Signature.empty*.

```
>>> 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'"
```

classmethod from_callable(*obj*, *, *follow_wrapped=True*)
Return a *Signature* (or its subclass) object for a given callable `obj`. Pass `follow_wrapped=False` to get a signature of `obj` without unwrapping its `__wrapped__` chain.

This method simplifies subclassing of *Signature*:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

버전 3.5에 추가.

class inspect.**Parameter**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)
Parameter objects are *immutable*. Instead of modifying a Parameter object, you can use *Parameter.replace()* to create a modified copy.

버전 3.5에서 변경: Parameter objects are picklable and hashable.

empty
A special class-level marker to specify absence of default values and annotations.

name
The name of the parameter as a string. The name must be a valid Python identifier.

CPython implementation detail: CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

버전 3.6에서 변경: These parameter names are exposed by this module as names like `implicit0`.

default
The default value for the parameter. If the parameter has no default value, this attribute is set to *Parameter.empty*.

annotation
The annotation for the parameter. If the parameter has no annotation, this attribute is set to *Parameter.empty*.

kind
Describes how argument values are bound to the parameter. Possible values (accessible via *Parameter*, like `Parameter.KEYWORD_ONLY`):

Name	Meaning
POSITIONAL_ONLY	Value must be supplied as a positional argument. Python has no explicit syntax for defining positional-only parameters, but many built-in and extension module functions (especially those that accept only one or two parameters) accept them.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
VAR_POSITIONAL	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.
VAR_KEYWORD	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```
>>> 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
```

replace(**[, name][, kind][, default][, annotation]*)

Create a new Parameter instance based on the instance replaced was invoked on. To override a *Parameter* attribute, pass the corresponding argument. To remove a default value or/and an annotation from a Parameter, pass *Parameter.empty*.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo:'spam'"
```

버전 3.4에서 변경: In Python 3.3 Parameter objects were allowed to have `name` set to `None` if their `kind`

was set to POSITIONAL_ONLY. This is no longer permitted.

class inspect.**BoundArguments**

Result of a *Signature.bind()* or *Signature.bind_partial()* call. Holds the mapping of arguments to the function's parameters.

arguments

An ordered, mutable mapping (*collections.OrderedDict*) of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in *arguments* will reflect in *args* and *kwargs*.

Should be used in conjunction with *Signature.parameters* for any argument processing purposes.

참고: Arguments for which *Signature.bind()* or *Signature.bind_partial()* relied on a default value are skipped. However, if needed, use *BoundArguments.apply_defaults()* to add them.

---

args

A tuple of positional arguments values. Dynamically computed from the *arguments* attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the *arguments* attribute.

signature

A reference to the parent *Signature* object.

apply_defaults()

Set default values for missing arguments.

For variable-positional arguments (*args) the default is an empty tuple.

For variable-keyword arguments (**kwargs) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

버전 3.5에 추가.

The *args* and *kwargs* properties can be used to invoke functions:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

더 보기:

PEP 362 - Function Signature Object. The detailed specification, implementation details and examples.

30.13.4 Classes and functions

inspect.**getclasstree**(*classes*, *unique=False*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

inspect.**getargspec**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* `ArgSpec(args, varargs, keywords, defaults)` is returned. *args* is a list of the parameter names. *varargs* and *keywords* are the names of the `*` and `**` parameters or `None`. *defaults* is a tuple of default argument values or `None` if there are no default arguments; if this tuple has *n* elements, they correspond to the last *n* elements listed in *args*.

버전 3.0부터 폐지: Use *`getfullargspec()`* for an updated API that is usually a drop-in replacement, but also correctly handles function annotations and keyword-only parameters.

Alternatively, use *`signature()`* and *Signature Object*, which provide a more structured introspection API for callables.

inspect.**getfullargspec**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* is returned:

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
annotations)
```

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names in declaration order. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that *`signature()`* and *Signature Object* provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

버전 3.4에서 변경: This function is now based on *`signature()`*, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

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: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

inspect.**getargvalues**(*frame*)

Get information about arguments passed into a particular frame. A *named tuple* `ArgInfo(args, varargs, keywords, locals)` is returned. *args* is a list of the argument names. *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

참고: This function was inadvertently marked as deprecated in Python 3.5.

---

inspect.**formatargspec**(*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])

Format a pretty argument spec from the values returned by *getfullargspec()*.

The first seven arguments are (`args`, `varargs`, `varkw`, `defaults`, `kwonlyargs`, `kwonlydefaults`, `annotations`).

The other six arguments are functions that are called to turn argument names, `*` argument name, `**` argument name, default values, return annotation and individual annotations into strings, respectively.

For example:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

버전 3.5부터 폐지: Use *signature()* and *Signature Object*, which provide a better introspecting API for callables.

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Format a pretty argument spec from the four values returned by *getargvalues()*. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

참고: This function was inadvertently marked as deprecated in Python 3.5.

---

inspect.**getmro**(*cls*)

Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

inspect.**getcallargs**(*func*, **args*, ***kwds*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> 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'
```

버전 3.2에 추가.

버전 3.5부터 폐지: Use *Signature.bind()* and *Signature.bind_partial()* instead.

inspect.**getclosurevars**(*func*)

Get the mapping of external name references in a Python function or method *func* to their current values. A *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function's module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

`TypeError` is raised if *func* is not a Python function or method.

버전 3.3에 추가.

inspect.**unwrap**(*func*, *, *stop=None*)

Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, `signature()` uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

`ValueError` is raised if a cycle is encountered.

버전 3.4에 추가.

30.13.5 The interpreter stack

When the following functions return "frame records," each record is a *named tuple* `FrameInfo(frame, filename, lineno, function, code_context, index)`. The tuple contains the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

버전 3.5에서 변경: Return a named tuple instead of a tuple.

참고: Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

inspect.**getframeinfo**(*frame*, *context=1*)
Get information about a frame or traceback object. A *named tuple* `Traceback(filename, lineno, function, code_context, index)` is returned.

inspect.**getouterframes**(*frame*, *context=1*)
Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

버전 3.5에서 변경: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

inspect.**getinnerframes**(*traceback*, *context=1*)
Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

버전 3.5에서 변경: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

inspect.**currentframe**()
Return the frame object for the caller's stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

inspect.**stack**(*context=1*)
Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

버전 3.5에서 변경: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

inspect.**trace**(*context=1*)
Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

버전 3.5에서 변경: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

30.13.6 Fetching attributes statically

Both *getattr()* and *hasattr()* can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. *getattr_static()* has the same signature as *getattr()* but avoids executing code when it fetches attributes.

inspect.**getattr_static**(*obj*, *attr*, *default=None*)
Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance *__dict__* is shadowed by another member (for example a property) then this function will be unable to find instance members.

버전 3.2에 추가.

getattr_static() does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
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:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

30.13.7 Current State of Generators and Coroutines

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. *getgeneratorstate()* allows the current state of a generator to be determined easily.

`inspect.`**`getgeneratorstate`**(*generator*)

Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.
- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

버전 3.2에 추가.

`inspect.`**`getcoroutinestate`**(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- CORO_CREATED: Waiting to start execution.
- CORO_RUNNING: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

버전 3.5에 추가.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

inspect.**getgeneratorlocals**(*generator*)
Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling *locals()* in the body of the generator, and all the same caveats apply.

If *generator* is a *generator* with no currently associated frame, then an empty dictionary is returned. *TypeError* is raised if *generator* is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

버전 3.3에 추가.

inspect.**getcoroutinelocals**(*coroutine*)
This function is analogous to *getgeneratorlocals()*, but works for coroutine objects created by `async def` functions.

버전 3.5에 추가.

30.13.8 Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

inspect.**CO_OPTIMIZED**
The code object is optimized, using fast locals.

inspect.**CO_NEWLOCALS**
If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

inspect.**CO_VARARGS**
The code object has a variable positional parameter (`*args`-like).

inspect.**CO_VARKEYWORDS**
The code object has a variable keyword parameter (`**kwargs`-like).

inspect.**CO_NESTED**
The flag is set when the code object is a nested function.

inspect.**CO_GENERATOR**
The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

inspect.**CO_NOFREE**
The flag is set if there are no free or cell variables.

inspect.**CO_COROUTINE**
The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** for more details.

버전 3.5에 추가.

inspect.**CO_ITERABLE_COROUTINE**
The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** for more details.

버전 3.5에 추가.

inspect.**CO_ASYNC_GENERATOR**
The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** for more details.

버전 3.6에 추가.

참고: The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the *inspect* module for any introspection needs.

30.13.9 Command Line Interface

The *inspect* module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

--details
Print information about the specified object rather than the source code

30.14 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.

-S를 사용하지 않는 한, 이 모듈의 임포트는 사이트별 경로를 모듈 검색 경로에 추가하고 몇 가지 내장(builtins)을 추가합니다. 사용되었다면, 이 모듈은 모듈 검색 경로를 자동으로 수정하거나 내장을 추가하지 않고도 안전하게 임포트 할 수 있습니다. 일반적인 사이트별 추가를 명시적으로 트리거 하려면, *site.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*/site-packages(유닉스와 맥에서)를 사용합니다. 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.

If a file named "pyvenv.cfg" exists one directory above sys.executable, sys.prefix and sys.exec_prefix are set to that directory and it is also checked for site-packages (sys.base_prefix and sys.base_exec_prefix will always be the "real" prefixes of the Python installation). If "pyvenv.cfg" (a bootstrap configuration file) contains the key "include-system-site-packages" set to anything other than "false" (case-insensitive), the system-level prefixes will still also 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.

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 spam이라는 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.

이러한 경로 조작 후, 임의의 사이트별 사용자 정의를 수행할 수 있는 sitecustomize라는 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, sitecustomize의 출력 시도는 무시됩니다. 다른 예외는 절차의 조용한 그리고 아마도 정체불명의 실패로 이어집니다.

그런 다음, *ENABLE_USER_SITE*가 참이면, 임의의 사용자별 사용자 정의를 수행할 수 있는 usercustomize라는 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 sitecustomize와 usercustomize 임포트는 여전히 시도됩니다.

30.14.1 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, sitecustomize나 usercustomize 모듈 또는 PYTHONSTARTUP 파일에서 *sys.__interactivehook__* 어트리뷰트를 삭제(또는 재정의)하십시오.

버전 3.4에서 변경: rlcompleter와 히스토리 활성화가 자동으로 이루어졌습니다.

30.14.2 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 X 빌드의 경우 ~/.local/lib/python*X.Y*/site-packages, 맥 프레임워크 빌드의 경우 ~/Library/Python/*X.Y*/lib/python/site-packages, 윈도우의 경우 *%APPDATA%*\Python\Python*XY*\site-packages입니다. 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 X 비 프레임워크 빌드의 경우 ~/.local, 맥 프레임워크 빌드의 경우 ~/Library/Python/*X.Y*, 윈도우의 경우 *%APPDATA%*\Python입니다. 이 값은 Distutils에서 사용자 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.path에 디렉터리를 추가하고 .pth 파일을 처리합니다. 일반적으로 sitecustomize나 usercustomize에서 사용됩니다(위를 참조하십시오).

site.**getsitepackages**()
모든 전역 site-packages 디렉터리를 포함하는 리스트를 반환합니다.

버전 3.2에 추가.

site.**getuserbase**()
사용자 베이스 디렉터리 *USER_BASE*의 경로를 반환합니다. 아직 초기화되지 않았으면, 이 함수는 PYTHONUSERBASE를 따라 설정합니다.

버전 3.2에 추가.

site.**getusersitepackages**()
사용자별 site-packages 디렉터리 *USER_SITE*의 경로를 반환합니다. 아직 초기화되지 않았으면, 이 함수는 PYTHONNOUSERSITE와 *USER_BASE*를 따라 설정합니다.

버전 3.2에 추가.

30.14.3 Command Line Interface

site 모듈은 명령 줄에서 사용자 디렉터리를 얻는 방법도 제공합니다:

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/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

CHAPTER 31

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:

31.1 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 딕셔너리를 지정합니다; 기본값은 키 `'__name__'`이 `'__console__'`로 설정되고 키 `'__doc__'`이 `None`으로 설정된 새로 만들어진 딕셔너리입니다.

class code.**InteractiveConsole**(*locals=None*, *filename="<console>"*)
대화형 파이썬 인터프리터의 동작을 가깝게 흉내 냅니다. 이 클래스는 *InteractiveInterpreter*를 기반으로 하며 친숙한 `sys.ps1`과 `sys.ps2`를 사용하는 프롬프트와 입력 버퍼링을 추가합니다.

code.**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*)
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. 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.

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 is the source string; *filename* is the optional filename from which source was read, defaulting to `'<input>'`; and *symbol* is the optional grammar start symbol, which should be `'single'` (the default), `'eval'` or `'exec'`.

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.

31.1.1 대화형 인터프리터 객체

InteractiveInterpreter.**runsource**(*source*, *filename="<input>"*, *symbol="single"*)
Compile and run some source in the interpreter. Arguments are the same as for *compile_command()*; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One of several things can happen:

- 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.

31.1.2 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 None이면, 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 True이고, 어떤 식으로든 줄이 처리되었으면 False입니다 (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.stdin에서 읽습니다; 서브 클래스는 이것을 다른 구현으로 바꿀 수 있습니다.

31.2 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 포함시키려면 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 되는데, 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.

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default), as a sequence of statements (`'exec'`) or as an *expression* (`'eval'`). Any other value will cause *ValueError* to be raised.

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 무시 될 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 32

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:

32.1 zipimport — Zip 저장소에서 모듈 임포트

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.

모든 파일이 ZIP 저장소에 있을 수 있지만, .py 와 .pyc 파일만 임포트 할 수 있습니다. 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.

저장소 주석이 포함된 ZIP 저장소는 현재 지원되지 않습니다.

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 273의 명세를 따르지만, Just van Rossum이 작성한 구현을 사용하는데 PEP 302에 설명된 임포트 훅을 사용합니다.

PEP 302 - 새 임포트 훅 이 모듈이 작동하는 데 도움이 되는 임포트 훅을 추가하는 PEP.

이 모듈은 예외를 정의합니다:

exception zipimport.**ZipImportError**
zipimporter 객체가 발생시키는 예외. *ImportError*의 서브 클래스이므로, *ImportError*로도 잡힐 수 있습니다.

32.1.1 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.

find_module(*fullname*[, *path*])
*fullname*로 지정된 모듈을 검색합니다. *fullname*은 완전히 정규화된 (점으로 구분된) 모듈 이름이어야 합니다. 모듈이 발견되면 zipimporter 인스턴스 자체를 반환하고, 그렇지 않으면 *None*을 반환합니다. 선택적 *path* 인자는 무시됩니다—임포터 프로토콜과의 호환성을 위해 있습니다.

get_code(*fullname*)
지정된 모듈의 코드 객체를 반환합니다. 모듈을 찾을 수 없으면 *ZipImportError*를 발생시킵니다.

get_data(*pathname*)
*pathname*와 관련된 데이터를 반환합니다. 파일을 찾을 수 없으면 *OSError*를 발생시킵니다.

버전 3.3에서 변경: *OSError* 대신 *IOError* 를 발생시켜왔습니다.

get_filename(*fullname*)
지정한 모듈이 임포트될 때 설정될 `__file__`의 값을 반환합니다. 모듈을 찾을 수 없으면 *ZipImportError*를 발생시킵니다.

버전 3.1에 추가.

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. 임포트 된 모듈을 반환하거나, 찾지 못하면 *ZipImportError*를 발생시킵니다.

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.

32.1.2 예제

다음은 ZIP 저장소에서 모듈을 임포트하는 예제입니다 - *zipimport* 모듈이 명시적으로 사용되지 않음에 유의하십시오.

```
$ unzip -l example.zip
Archive:  example.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

32.2 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.

버전 3.6에 추가.

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__)
```

그러면 `sys.path`에 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.

name 인자와 일치하는 `*.pkg` 파일도 찾습니다. 이 기능은 `import`로 시작하는 줄을 특수하게 다루지 않는다는 점을 제외하면, `*.pth` 파일과 유사합니다 (자세한 내용은 *site* 모듈을 참조하십시오). `*.pkg` 파일을 액면 그대로 신뢰합니다: 중복은 확인하지만, `*.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.path*가 시퀀스라고 가정합니다. 존재하는 디렉터리를 참조하는 문자열이 아닌 *sys.path* 항목은 무시됩니다. 파일명으로 사용될 때 에러를 일으키는 *sys.path*의 유니코드 항목은 이 함수가 예외를 발생시키도록 할 수 있습니다 (*os.path.isdir()* 동작과 일치합니다).

class pkgutil.**ImpImporter**(*dirname=None*)
파이썬의 "고전적인" 임포트 알고리즘을 감싸는 **PEP 302** 파인더.

*dirname*이 문자열이면, 해당 디렉터리를 검색하는 **PEP 302** 파인더가 만들어집니다. *dirname*이 None이면, 현재 *sys.path*와 프로즌 또는 내장 모듈을 검색하는 **PEP 302** 파인더가 만들어집니다.

*ImpImporter*는 현재 *sys.meta_path*에 넣어서 사용하는 것을 지원하지 않음에 유의하십시오.

버전 3.3부터 폐지: 표준 임포트 메커니즘이 이제 완전히 PEP 302를 준수하고 *importlib*에서 사용할 수 있으므로, 이 에뮬레이션은 더는 필요하지 않습니다.

class pkgutil.**ImpLoader**(*fullname*, *file*, *filename*, *etc*)
파이썬의 "고전적인" 임포트 알고리즘을 감싸는 로더.

버전 3.3부터 폐지: 표준 임포트 메커니즘이 이제 완전히 PEP 302를 준수하고 *importlib*에서 사용할 수 있으므로, 이 에뮬레이션은 더는 필요하지 않습니다.

pkgutil.**find_loader**(*fullname*)
주어진 *fullname*에 대한 모듈 로더를 가져옵니다.

이것은 *importlib.util.find_spec()*을 감싸는 하위 호환성 래퍼인데, 대부분의 실패를 *ImportError*로 변환하고 전체 ModuleSpec이 아닌 로더만 반환합니다.

버전 3.3에서 변경: 패키지 내부 PEP 302 임포트 에뮬레이션에 의존하는 대신, *importlib*에 직접 기반하도록 갱신되었습니다.

버전 3.4에서 변경: **PEP 451**에 기반하도록 갱신되었습니다

pkgutil.**get_importer**(*path_item*)
주어진 *path_item*에 대한 파인더를 가져옵니다.

반환된 파인더는 경로 훅 때문에 새로 만들어지면 *sys.path_importer_cache*에 캐시 됩니다.

*sys.path_hooks*의 재검색이 필요하면, 캐시(또는 그 일부)를 수동으로 지울 수 있습니다.

버전 3.3에서 변경: 패키지 내부 PEP 302 임포트 에뮬레이션에 의존하는 대신, *importlib*에 직접 기반하도록 갱신되었습니다.

pkgutil.**get_loader**(*module_or_name*)
*module_or_name*에 대한 로더 객체를 가져옵니다.

모듈이나 패키지가 일반 임포트 메커니즘을 통해 액세스할 수 있으면, 그 장치의 관련 부분을 감싸는 래퍼가 반환됩니다. 모듈을 찾거나 임포트 할 수 없으면 None을 반환합니다. 명명된 모듈이 아직 임포트 되지 않았다면, 패키지 __path__를 구성하기 위해 포함하는 패키지(있다면))를 임포트 합니다.

버전 3.3에서 변경: 패키지 내부 PEP 302 임포트 에뮬레이션에 의존하는 대신, *importlib*에 직접 기반하도록 갱신되었습니다.

버전 3.4에서 변경: **PEP 451**에 기반하도록 갱신되었습니다

pkgutil.**iter_importers**(*fullname=''*)
주어진 모듈 이름에 대해 파인더 객체를 산출(yield)합니다.

fullname에 '.'이 포함되어 있으면, 파인더는 fullname을 포함하는 패키지를 위한 것입니다, 그렇지 않으면, 등록된 모든 최상위 수준 파인더입니다 (즉, sys.meta_path 와 sys.path_hooks에 있는 것들).

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, *importlib*에 직접 기반하도록 갱신되었습니다.

pkgutil.**iter_modules**(*path=None*, *prefix=''*)
*path*의 모든 서브 모듈에 대한 *ModuleInfo*를, 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.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.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)
*path*에 재귀적으로 포함된 모든 모듈이나, *path*가 None이면 모든 액세스할 수 있는 모듈에 대한 *ModuleInfo*를 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:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
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, None이 반환됩니다. 특히, 이름 공간 패키지를 위한 로더는 *get_data*를 지원하지 않습니다.

32.3 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.py는 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.path가 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.

32.3.1 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__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
----------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

32.4 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 모듈 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. 제공된 딕셔너리는 수정되지 않습니다. 아래의 특수 전역 변수가 제공된 딕셔너리에 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.

runpy.**run_path**(*file_path*, *init_globals=None*, *run_name=None*)

명명된 파일 시스템 위치에 있는 코드를 실행하고 결과 모듈 전역 딕셔너리를 반환합니다. CPython 명령 줄에 제공된 스크립트 이름과 마찬가지로, 제공된 경로는 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. 제공된 딕셔너리는 수정되지 않습니다. 아래의 특수 전역 변수가 제공된 딕셔너리에 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 *None*이 아니면) *run_name*으로, 그렇지 않으면 '<run_path>'로 설정됩니다.

제공된 경로가 스크립트 파일(소스나 사전 컴파일된 바이트 코드)을 직접 참조하면, __file__은 제공된 경로로 설정되고 __spec__, __cached__, __loader__ 및 __package__는 모두 *None*으로 설정됩니다.

제공된 경로가 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 *sys*의 항목에 대한 모든 수정 내용이 되돌려집니다.

*run_module()*과 달리, *sys*에 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).

버전 3.2에 추가.

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.

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

32.5 importlib — The implementation of import

버전 3.1에 추가.

Source code: Lib/importlib/__init__.py

32.5.1 Introduction

The purpose of the *importlib* package is two-fold. One is to provide the implementation of the `import` statement (and thus, by extension, the *__import__()* function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

더 보기:

import The language reference for the `import` statement.

Packages specification Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in *sys.modules*).

The *__import__()* function The `import` statement is syntactic sugar for this function.

PEP 235 Import on Case-Insensitive Platforms

PEP 263 Defining Python Source Code Encodings

PEP 302 New Import Hooks

PEP 328 Imports: Multi-Line and Absolute/Relative

PEP 366 Main module explicit relative imports

PEP 420 Implicit namespace packages

PEP 451 A ModuleSpec Type for the Import System

PEP 488 Elimination of PYO files

PEP 489 Multi-phase extension module initialization

PEP 552 Deterministic pycs

PEP 3120 Using UTF-8 as the Default Source Encoding

PEP 3147 PYC Repository Directories

32.5.2 Functions

`importlib.__import__`(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
An implementation of the built-in *__import__()* function.

> **참고:** Programmatic importing of modules should use *import_module()* instead of this function.

`importlib.import_module`(*name*, *package=None*)
Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The *import_module()* function acts as a simplifying wrapper around *importlib.__import__()*. This means all semantics of the function are derived from *importlib.__import__()*. The most important difference between these two functions is that *import_module()* returns the specified package or module (e.g. `pkg.mod`), while *__import__()* returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call *invalidate_caches()* in order for the new module to be noticed by the import system.

버전 3.3에서 변경: Parent packages are automatically imported.

importlib.**find_loader**(*name*, *path=None*)

Find the loader for a module, optionally within the specified *path*. If the module is in *sys.modules*, then sys.modules[name].__loader__ is returned (unless the loader would be None or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. None is returned if no loader is found.

A dotted name does not have its parents implicitly imported as that requires loading them and that may not be desired. To properly import a submodule you will need to import all parent packages of the submodule and use the correct argument to *path*.

버전 3.3에 추가.

버전 3.4에서 변경: If __loader__ is not set, raise *ValueError*, just like when the attribute is set to None.

버전 3.4부터 폐지: Use *importlib.util.find_spec()* instead.

importlib.**invalidate_caches**()

Invalidate the internal caches of finders stored at *sys.meta_path*. If a finder implements invalidate_caches() then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

버전 3.3에 추가.

importlib.**reload**(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in *sys.modules*).

When *reload()* is executed:

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. The init function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a try statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading *sys*, *__main__*, *builtins* and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` … `import` …, calling *reload()* for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

버전 3.4에 추가.

버전 3.7에서 변경: *ModuleNotFoundError* is raised when the module being reloaded lacks a `ModuleSpec`.

32.5.3 `importlib.abc` – Abstract base classes related to import

Source code: Lib/importlib/abc.py

The *importlib.abc* module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy:

```
object
 +-- Finder (deprecated)
 |    +-- MetaPathFinder
 |    +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**Finder**
An abstract base class representing a *finder*.

버전 3.3부터 폐지: Use *MetaPathFinder* or *PathEntryFinder* instead.

abstractmethod find_module(*fullname*, *path=None*)
An abstract method for finding a *loader* for the specified module. Originally specified in **PEP 302**, this method was meant for use in *sys.meta_path* and in the path-based import subsystem.

버전 3.4에서 변경: Returns `None` when called instead of raising *NotImplementedError*.

class importlib.abc.**MetaPathFinder**
An abstract base class representing a *meta path finder*. For compatibility, this is a subclass of *Finder*.

버전 3.3에 추가.

find_spec(*fullname*, *path*, *target=None*)
An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete `MetaPathFinders`.

버전 3.4에 추가.

find_module(*fullname*, *path*)
A legacy method for finding a *loader* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a loader cannot be found, `None` is returned.

If *find_spec()* is defined, backwards-compatible functionality is provided.

버전 3.4에서 변경: Returns `None` when called instead of raising *NotImplementedError*. Can use *find_spec()* to provide functionality.

버전 3.4부터 폐지: Use *find_spec()* instead.

invalidate_caches()
An optional method which, when called, should invalidate any internal cache used by the finder. Used by *importlib.invalidate_caches()* when invalidating the caches of all finders on *sys.meta_path*.

버전 3.4에서 변경: Returns `None` when called instead of `NotImplemented`.

class importlib.abc.**PathEntryFinder**
An abstract base class representing a *path entry finder*. Though it bears some similarities to *MetaPathFinder*, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by `PathFinder`. This ABC is a subclass of *Finder* for compatibility reasons only.

버전 3.3에 추가.

find_spec(*fullname*, *target=None*)
An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete `PathEntryFinders`.

버전 3.4에 추가.

find_loader(*fullname*)
A legacy method for finding a *loader* for the specified module. Returns a 2-tuple of `(loader, portion)` where `portion` is a sequence of file system locations contributing to part of a namespace package. The loader may be `None` while specifying `portion` to signify the contribution of the file system locations to a namespace package. An empty list can be used for `portion` to signify the loader is not part of a namespace package. If `loader` is `None` and `portion` is the empty list then no loader or location for a namespace package were found (i.e. failure to find anything for the module).

If *find_spec()* is defined then backwards-compatible functionality is provided.

버전 3.4에서 변경: Returns `(None, [])` instead of raising *NotImplementedError*. Uses *find_spec()* when available to provide functionality.

버전 3.4부터 폐지: Use *find_spec()* instead.

find_module(*fullname*)
A concrete implementation of *Finder.find_module()* which is equivalent to `self.find_loader(fullname)[0]`.

버전 3.4부터 폐지: Use *find_spec()* instead.

invalidate_caches()
An optional method which, when called, should invalidate any internal cache used by the finder. Used by `PathFinder.invalidate_caches()` when invalidating the caches of all cached finders.

class importlib.abc.**Loader**

An abstract base class for a *loader*. See **PEP 302** for the exact definition for a loader.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.abc.ResourceReader*.

버전 3.7에서 변경: Introduced the optional `get_resource_reader()` method.

create_module(*spec*)

A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

버전 3.4에 추가.

버전 3.5에서 변경: Starting in Python 3.6, this method will not be optional when *exec_module()* is defined.

exec_module(*module*)

An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when `exec_module()` is called. When this method exists, *create_module()* must be defined.

버전 3.4에 추가.

버전 3.6에서 변경: *create_module()* must also be defined.

load_module(*fullname*)

A legacy method for loading a module. If the module cannot be loaded, *ImportError* is raised, otherwise the loaded module is returned.

If the requested module already exists in *sys.modules*, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into *sys.modules* before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from *sys.modules*; modules already in *sys.modules* before the loader began execution should be left alone (see *importlib.util.module_for_loader()*).

The loader should set several attributes on the module. (Note that some of these attributes can change when a module is reloaded):

- **__name__** The name of the module.
- **__file__** The path to where the module data is stored (not set for built-in modules).
- **__cached__** The path to where a compiled version of the module is/should be stored (not set when the attribute would be inappropriate).
- **__path__** A list of strings specifying the search path within a package. This attribute is not set on modules.
- **__package__** The parent package for the module/package. If the module is top-level then it has a value of the empty string. The *importlib.util.module_for_loader()* decorator can handle the details for __package__.
- **__loader__** The loader used to load the module. The *importlib.util.module_for_loader()* decorator can handle the details for __package__.

When *exec_module()* is available then backwards-compatible functionality is provided.

버전 3.4에서 변경: Raise *ImportError* when called instead of *NotImplementedError*. Functionality provided when *exec_module()* is available.

버전 3.4부터 폐지: The recommended API for loading a module is *exec_module()* (and *create_module()*). Loaders should implement it instead of load_module(). The import machinery takes care of all the other responsibilities of load_module() when exec_module() is implemented.

module_repr(*module*)
A legacy method which when implemented calculates and returns the given module's repr, as a string. The module type's default repr() will use the result of this method as appropriate.

버전 3.3에 추가.

버전 3.4에서 변경: Made optional instead of an abstractmethod.

버전 3.4부터 폐지: The import machinery now takes care of this automatically.

class importlib.abc.**ResourceReader**
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 in a e.g. 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 3.7에 추가.

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.abc.**ResourceLoader**
An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

버전 3.7부터 폐지: This ABC is deprecated in favour of supporting resource loading through `importlib.abc.ResourceReader`.

abstractmethod get_data(*path*)
An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. *OSError* is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

버전 3.4에서 변경: Raises *OSError* instead of *NotImplementedError*.

class importlib.abc.**InspectLoader**
An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

get_code(*fullname*)
Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an *ImportError* if loader cannot find the requested module.

참고: While the method has a default implementation, it is suggested that it be overridden if possible for performance.

버전 3.4에서 변경: No longer abstract and a concrete implementation is provided.

abstractmethod get_source(*fullname*)
An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises *ImportError* if the loader cannot find the module specified.

버전 3.4에서 변경: Raises *ImportError* instead of *NotImplementedError*.

is_package(*fullname*)
An abstract method to return a true value if the module is a package, a false value otherwise. *ImportError* is raised if the *loader* cannot find the module.

버전 3.4에서 변경: Raises *ImportError* instead of *NotImplementedError*.

static source_to_code(*data*, *path='<string>'*)
Create a code object from Python source.

The *data* argument can be whatever the *compile()* function supports (i.e. string or bytes). The *path* argument should be the "path" to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

버전 3.4에 추가.

버전 3.5에서 변경: Made the method static.

exec_module(*module*)
Implementation of *Loader.exec_module()*.

버전 3.4에 추가.

load_module(*fullname*)
Implementation of *Loader.load_module()*.

버전 3.4부터 폐지: use *exec_module()* instead.

class importlib.abc.**ExecutionLoader**
An abstract base class which inherits from *InspectLoader* that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

abstractmethod get_filename(*fullname*)
An abstract method that is to return the value of `__file__` for the specified module. If no path is available, *ImportError* is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

버전 3.4에서 변경: Raises *ImportError* instead of *NotImplementedError*.

class importlib.abc.**FileLoader**(*fullname*, *path*)
An abstract base class which inherits from *ResourceLoader* and *ExecutionLoader*, providing concrete implementations of *ResourceLoader.get_data()* and *ExecutionLoader.get_filename()*.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

버전 3.3에 추가.

name
The name of the module the loader can handle.

path
Path to the file of the module.

load_module(*fullname*)
Calls super's `load_module()`.

버전 3.4부터 폐지: Use *Loader.exec_module()* instead.

abstractmethod get_filename(*fullname*)
Returns *path*.

abstractmethod get_data(*path*)
Reads *path* as a binary file and returns the bytes from it.

class importlib.abc.**SourceLoader**
An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both *ResourceLoader* and *ExecutionLoader*, requiring the implementation of:

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()*** Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise *NotImplementedError*) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats(*path*)
Optional abstract method which returns a *dict* containing metadata about the specified path. Supported dictionary keys are:

- `'mtime'` (mandatory): an integer or floating-point number representing the modification time of the source code;
- `'size'` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, *OSError* is raised.

버전 3.3에 추가.

버전 3.4에서 변경: Raise *OSError* instead of *NotImplementedError*.

path_mtime(*path*)
Optional abstract method which returns the modification time for the specified path.

버전 3.3부터 폐지: This method is deprecated in favour of *path_stats()*. You don't have to implement it, but it is still available for compatibility purposes. Raise *OSError* if the path cannot be handled.

버전 3.4에서 변경: Raise *OSError* instead of *NotImplementedError*.

set_data(*path*, *data*)
Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (*errno.EACCES*/*PermissionError*), do not propagate the exception.

버전 3.4에서 변경: No longer raises *NotImplementedError* when called.

get_code(*fullname*)
Concrete implementation of *InspectLoader.get_code()*.

exec_module(*module*)
Concrete implementation of *Loader.exec_module()*.

버전 3.4에 추가.

load_module(*fullname*)
Concrete implementation of *Loader.load_module()*.

버전 3.4부터 폐지: Use *exec_module()* instead.

get_source(*fullname*)
Concrete implementation of *InspectLoader.get_source()*.

is_package(*fullname*)
Concrete implementation of *InspectLoader.is_package()*. A module is determined to be a package if its file path (as provided by *ExecutionLoader.get_filename()*) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

32.5.4 importlib.resources – Resources

Source code: Lib/importlib/resources.py

버전 3.7에 추가.

This module leverages Python's import system to provide access to *resources* within *packages*. If you can import a package, you can access resources within that package. 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.

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.abc.ResourceReader*.

The following types are defined.

`importlib.resources.`**`Package`**
: The `Package` type is defined as `Union[str, ModuleType]`. This means that where the function describes accepting a `Package`, you can pass in either a string or a module. Module objects must have a resolvable `__spec__.submodule_search_locations` that is not `None`.

`importlib.resources.`**`Resource`**
: This type describes the resource names passed into the various functions in this package. This is defined as `Union[str, os.PathLike]`.

The following functions are available.

`importlib.resources.`**`open_binary`**(*package*, *resource*)
: Open for binary reading the *resource* within *package*.

  package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns a `typing.BinaryIO` instance, a binary I/O stream open for reading.

`importlib.resources.`**`open_text`**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
: Open for text reading the *resource* within *package*. By default, the resource is opened for reading as UTF-8.

  package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in *open()*.

  This function returns a `typing.TextIO` instance, a text I/O stream open for reading.

`importlib.resources.`**`read_binary`**(*package*, *resource*)
: Read and return the contents of the *resource* within *package* as `bytes`.

  package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns the contents of the resource as *bytes*.

`importlib.resources.`**`read_text`**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
: Read and return the contents of *resource* within *package* as a `str`. By default, the contents are read as strict UTF-8.

  package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in *open()*. This function returns the contents of the resource as *str*.

`importlib.resources.`**`path`**(*package*, *resource*)
: Return 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 file created when the resource needs to be extracted from e.g. a zip file.

  package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory).

`importlib.resources.`**`is_resource`**(*package*, *name*)
: Return `True` if there is a resource named *name* in the package, otherwise `False`. Remember that directories are *not* resources! *package* is either a name or a module object which conforms to the `Package` requirements.

importlib.resources.**contents**(*package*)
Return an iterable over the named items within the package. The iterable returns *str* resources (e.g. files) and non-resources (e.g. directories). The iterable does not recurse into subdirectories.

package is either a name or a module object which conforms to the `Package` requirements.

32.5.5 importlib.machinery – Importers and path hooks

Source code: Lib/importlib/machinery.py

This module contains the various objects that help `import` find and load modules.

importlib.machinery.**SOURCE_SUFFIXES**
A list of strings representing the recognized file suffixes for source modules.

버전 3.3에 추가.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**
A list of strings representing the file suffixes for non-optimized bytecode modules.

버전 3.3에 추가.

버전 3.5부터 폐지: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**
A list of strings representing the file suffixes for optimized bytecode modules.

버전 3.3에 추가.

버전 3.5부터 폐지: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**BYTECODE_SUFFIXES**
A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

버전 3.3에 추가.

버전 3.5에서 변경: The value is no longer dependent on `__debug__`.

importlib.machinery.**EXTENSION_SUFFIXES**
A list of strings representing the recognized file suffixes for extension modules.

버전 3.3에 추가.

importlib.machinery.**all_suffixes**()
Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, *inspect.getmodulename()*).

버전 3.3에 추가.

class importlib.machinery.**BuiltinImporter**
An *importer* for built-in modules. All known built-in modules are listed in *sys.builtin_module_names*. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

버전 3.5에서 변경: As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

class importlib.machinery.**FrozenImporter**
An *importer* for frozen modules. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

버전 3.4에서 변경: Gained `create_module()` and `exec_module()` methods.

class importlib.machinery.**WindowsRegistryFinder**
Finder for modules declared in the Windows registry. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

버전 3.3에 추가.

버전 3.6부터 폐지: Use *site* configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.**PathFinder**
A *Finder* for *sys.path* and package `__path__` attributes. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(*fullname*, *path=None*, *target=None*)
Class method that attempts to find a *spec* for the module specified by *fullname* on *sys.path* or, if defined, on *path*. For each path entry that is searched, *sys.path_importer_cache* is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in *sys.path_importer_cache*, then *sys.path_hooks* is searched for a finder for the path entry and, if found, is stored in *sys.path_importer_cache* along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

버전 3.4에 추가.

버전 3.5에서 변경: If the current working directory – represented by an empty string – is no longer valid then `None` is returned but no value is cached in *sys.path_importer_cache*.

classmethod find_module(*fullname*, *path=None*)
A legacy wrapper around *find_spec()*.

버전 3.4부터 폐지: Use *find_spec()* instead.

classmethod invalidate_caches()
Calls *importlib.abc.PathEntryFinder.invalidate_caches()* on all finders stored in *sys.path_importer_cache* that define the method. Otherwise entries in *sys.path_importer_cache* set to `None` are deleted.

버전 3.7에서 변경: Entries of `None` in *sys.path_importer_cache* are deleted.

버전 3.4에서 변경: Calls objects in *sys.path_hooks* with the current working directory for `''` (i.e. the empty string).

class importlib.machinery.**FileFinder**(*path*, **loader_details*)
A concrete implementation of *importlib.abc.PathEntryFinder* which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call `importlib.invalidate_caches()`.

버전 3.3에 추가.

path
The path the finder will search in.

find_spec(*fullname*, *target=None*)
Attempt to find the spec to handle *fullname* within `path`.

버전 3.4에 추가.

find_loader(*fullname*)
Attempt to find the loader to handle *fullname* within `path`.

invalidate_caches()
Clear out the internal cache.

classmethod path_hook(**loader_details*)
A class method which returns a closure for use on `sys.path_hooks`. An instance of `FileFinder` is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, `ImportError` is raised.

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)
A concrete implementation of `importlib.abc.SourceLoader` by subclassing `importlib.abc.FileLoader` and providing some concrete implementations of other methods.

버전 3.3에 추가.

name
The name of the module that this loader will handle.

path
The path to the source file.

is_package(*fullname*)
Return `True` if `path` appears to be for a package.

path_stats(*path*)
Concrete implementation of `importlib.abc.SourceLoader.path_stats()`.

set_data(*path*, *data*)
Concrete implementation of `importlib.abc.SourceLoader.set_data()`.

load_module(*name=None*)
Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

버전 3.6부터 폐지: Use `importlib.abc.Loader.exec_module()` instead.

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)
A concrete implementation of `importlib.abc.FileLoader` which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

버전 3.3에 추가.

name
The name of the module the loader will handle.

path
The path to the bytecode file.

is_package(*fullname*)
Determines if the module is a package based on *path*.

get_code(*fullname*)
Returns the code object for *name* created from *path*.

get_source(*fullname*)
Returns `None` as bytecode files have no source when this loader is used.

load_module(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

버전 3.6부터 폐지: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)
A concrete implementation of *importlib.abc.ExecutionLoader* for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

버전 3.3에 추가.

name
Name of the module the loader supports.

path
Path to the extension module.

create_module(*spec*)
Creates the module object from the given specification in accordance with **PEP 489**.

버전 3.5에 추가.

exec_module(*module*)
Initializes the given module object in accordance with **PEP 489**.

버전 3.5에 추가.

is_package(*fullname*)
Returns `True` if the file path points to a package's `__init__` module based on *EXTENSION_SUFFIXES*.

get_code(*fullname*)
Returns `None` as extension modules lack a code object.

get_source(*fullname*)
Returns `None` as extension modules do not have source code.

get_filename(*fullname*)
Returns *path*.

버전 3.4에 추가.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, *, *origin=None*, *loader_state=None*, *is_package=None*)
A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. In the descriptions below, the names in parentheses give the corresponding attribute available directly on the module object. E.g. `module.__spec__.origin == module.__file__`. Note however that

while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. Thus it is possible to update the module's `__path__` at runtime, and this will not be automatically reflected in `__spec__.submodule_search_locations`.

버전 3.4에 추가.

name

(`__name__`)

A string for the fully-qualified name of the module.

loader

(`__loader__`)

The loader to use for loading. For namespace packages this should be set to `None`.

origin

(`__file__`)

Name of the place from which the module is loaded, e.g. "builtin" for built-in modules and the filename for modules loaded from source. Normally "origin" should be set, but it may be `None` (the default) which indicates it is unspecified (e.g. for namespace packages).

submodule_search_locations

(`__path__`)

List of strings for where to find submodules, if a package (`None` otherwise).

loader_state

Container of extra module-specific data for use during loading (or `None`).

cached

(`__cached__`)

String for where the compiled module should be stored (or `None`).

parent

(`__package__`)

(Read-only) Fully-qualified name of the package to which the module belongs as a submodule (or `None`).

has_location

Boolean indicating whether or not the module's "origin" attribute refers to a loadable location.

32.5.6 `importlib.util` – Utility code for importers

Source code: Lib/importlib/util.py

This module contains the various objects that help in the construction of an *importer*.

`importlib.util.`**`MAGIC_NUMBER`**

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider *`importlib.abc.SourceLoader`*.

버전 3.4에 추가.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, *, *optimization=None*)

Return the **PEP 3147**/**PEP 488** path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`; if `sys.implementation.cache_tag` is not defined then *NotImplementedError* will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `''` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` causes the interpter's optimization level to be used. Any other value's string representation being used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else *ValueError* is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* an *optimization* are not `None` then *TypeError* is raised.

버전 3.4에 추가.

버전 3.5에서 변경: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

버전 3.6에서 변경: Accepts a *path-like object*.

importlib.util.**source_from_cache**(*path*)

Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** or **PEP 488** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

버전 3.4에 추가.

버전 3.6에서 변경: Accepts a *path-like object*.

importlib.util.**decode_source**(*source_bytes*)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by *importlib.abc.InspectLoader.get_source()*).

버전 3.4에 추가.

importlib.util.**resolve_name**(*name*, *package*)

Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __package__)` without doing a check to see if the **package** argument is needed.

ValueError is raised if **name** is a relative module name but package is a false value (e.g. `None` or the empty string). *ValueError* is also raised a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

버전 3.3에 추가.

importlib.util.**find_spec**(*name*, *package=None*)

Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in *sys.modules*, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

버전 3.4에 추가.

버전 3.7에서 변경: Raises *ModuleNotFoundError* instead of *AttributeError* if **package** is in fact not a package (i.e. lacks a __path__ attribute).

importlib.util.**module_from_spec**(*spec*)

Create a new module based on **spec** and *spec.loader.create_module*.

If *spec.loader.create_module* does not return None, then any pre-existing attributes will not be reset. Also, no *AttributeError* will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using *types.ModuleType* to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

버전 3.5에 추가.

@importlib.util.**module_for_loader**

A *decorator* for *importlib.abc.Loader.load_module()* to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. load_module(self, module)) for which the second argument will be the module **object** to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.

The decorated method will take in the **name** of the module to be loaded as expected for a *loader*. If the module is not found in *sys.modules* then a new one is constructed. Regardless of where the module came from, __loader__ set to **self** and __package__ is set based on what *importlib.abc.InspectLoader.is_package()* returns (if available). These attributes are set unconditionally to support reloading.

If an exception is raised by the decorated method and a module was added to *sys.modules*, then the module will be removed to prevent a partially initialized module from being in left in *sys.modules*. If the module was already in *sys.modules* then it is left alone.

버전 3.3에서 변경: __loader__ and __package__ are automatically set (when possible).

버전 3.4에서 변경: Set __name__, __loader__ __package__ unconditionally to support reloading.

버전 3.4부터 폐지: The import machinery now directly performs all the functionality provided by this function.

@importlib.util.**set_loader**

A *decorator* for *importlib.abc.Loader.load_module()* to set the __loader__ attribute on the returned module. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. self) is what __loader__ should be set to.

버전 3.4에서 변경: Set __loader__ if set to None, as if the attribute does not exist.

버전 3.4부터 폐지: The import machinery takes care of this automatically.

@importlib.util.**set_package**

A *decorator* for *importlib.abc.Loader.load_module()* to set the __package__ attribute on the returned module. If __package__ is set and has a value other than None it will not be changed.

버전 3.4부터 폐지: The import machinery takes care of this automatically.

importlib.util.**spec_from_loader**(*name*, *loader*, ***, *origin=None*, *is_package=None*)

A factory function for creating a ModuleSpec instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available *loader* APIs, such as InspectLoader.is_package(), to fill in any missing information on the spec.

버전 3.4에 추가.

importlib.util.**spec_from_file_location**(*name*, *location*, ***, *loader=None*, *submodule_search_locations=None*)

A factory function for creating a ModuleSpec instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

버전 3.4에 추가.

버전 3.6에서 변경: Accepts a *path-like object*.

`importlib.util.`**`source_hash`**(*source_bytes*)
Return the hash of *source_bytes* as bytes. A hash-based `.pyc` file embeds the *source_hash()* of the corresponding source file's contents in its header.

버전 3.7에 추가.

class `importlib.util.`**`LazyLoader`**(*loader*)
A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define *exec_module()* as control over what module type is used for the module is required. For those same reasons, the loader's *create_module()* method must return `None` or a type for which its `__class__` attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into *sys.modules* will not work as there is no way to properly replace the module references throughout the interpreter safely; *ValueError* is raised if such a substitution is detected.

참고: For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

버전 3.5에 추가.

버전 3.6에서 변경: Began calling *create_module()*, removing the compatibility warning for *importlib.machinery.BuiltinImporter* and *importlib.machinery.ExtensionFileLoader*.

classmethod **`factory`**(*loader*)
A static method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

32.5.7 Examples

Importing programmatically

To programmatically import a module, use *importlib.import_module()*.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use *importlib.util.find_spec()*.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

spec = importlib.util.find_spec(name)
if spec is None:
    print("can't find the itertools module")
else:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    # Adding the module to sys.modules is optional.
    sys.modules[name] = module
```

Importing a source file directly

To import a Python source file directly, use the following recipe (Python 3.5 and newer only):

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(module)
# Optional; only necessary if you want to be able to import the module
# by name later.
sys.modules[module_name] = module
```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on *sys.meta_path* while the latter is what you create using a *path entry hook* on *sys.path_hooks* which works with *sys.path* entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of *importlib.import_module()* (Python 3.4 and newer for the importlib usage, Python 3.6 and newer for other parts of the code).

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    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)
    spec.loader.exec_module(module)
    sys.modules[absolute_name] = module
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

CHAPTER 33

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:

33.1 `parser` — Access Python parse trees

The *parser* module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

참고: From Python 2.5 onward, it's much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the *ast* module.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the *parser* module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to reference-index. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the *expr()* or *suite()* functions, described below. The ST objects created by *sequence2st()* faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, though source code has usually been forward-compatible within a major release series.

Each element of the sequences returned by *st2list()* or *st2tuple()* has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module *symbol*. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module *token*.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple "wrapper" class may be created in Python to hide the use of ST objects.

The *parser* module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

더 보기:

Module *symbol* Useful constants representing internal nodes of the parse tree.

Module *token* Useful constants representing leaf nodes of the parse tree and functions for testing node values.

33.1.1 Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `'eval'` and `'exec'` forms.

`parser.`**`expr`**(*source*)
: The *expr()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`suite`**(*source*)
: The *suite()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`sequence2st`**(*sequence*)
: This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a *ParserError* exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to *compilest()*. This may indicate problems not related to syntax (such as a *MemoryError* exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

  Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a

valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

parser.**tuple2st**(*sequence*)
This is the same function as *sequence2st()*. This entry point is maintained for backward compatibility.

33.1.2 Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple-trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

parser.**st2list**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, *st2tuple()* should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

parser.**st2tuple**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to *st2list()*.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

parser.**compilest**(*st*, *filename='<syntax-tree>'*)
The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in *exec()* or *eval()* functions. This function provides the interface to the compiler, passing the internal parse tree from *st* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a *SyntaxError* caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The *SyntaxError* raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the *parser* module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

33.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via *expr()* or *suite()* or from a parse tree via *sequence2st()*.

parser.**isexpr**(*st*)
When *st* represents an `'eval'` form, this function returns `True`, otherwise it returns `False`. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by *compilest()* cannot be queried like this either, and are identical to those created by the built-in *compile()* function.

parser.**issuite**(*st*)
This function mirrors *isexpr()* in that it reports whether an ST object represents an `'exec'` form, commonly

known as a "suite." It is not safe to assume that this function is equivalent to `not isexpr(st)`, as additional syntactic fragments may be supported in the future.

33.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception parser.**ParserError**
Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in *SyntaxError* raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to *sequence2st()* and an explanatory string. Calls to *sequence2st()* need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions *compilest()*, *expr()*, and *suite()* may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions *MemoryError*, *OverflowError*, *SyntaxError*, and *SystemError*. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

33.1.5 ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the *pickle* module) is also supported.

parser.**STType**
The type of the objects returned by *expr()*, *suite()* and *sequence2st()*.

ST objects have the following methods:

ST.**compile**(*filename='<syntax-tree>'*)
Same as `compilest(st, filename)`.

ST.**isexpr**()
Same as `isexpr(st)`.

ST.**issuite**()
Same as `issuite(st)`.

ST.**tolist**(*line_info=False*, *col_info=False*)
Same as `st2list(st, line_info, col_info)`.

ST.**totuple**(*line_info=False*, *col_info=False*)
Same as `st2tuple(st, line_info, col_info)`.

33.1.6 Example: Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the *parser* module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the *parser* module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

33.2 `ast` — Abstract Syntax Trees

Source code: Lib/ast.py

The *ast* module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing `ast.PyCF_ONLY_AST` as a flag to the *compile()* built-in function, or using the *parse()* helper provided in this module. The result will be a tree of objects whose classes all inherit from *ast.AST*. An abstract syntax tree can be compiled into a Python code object using the built-in *compile()* function.

33.2.1 Node classes

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 *below*. They are defined in the `_ast` C module and re-exported in *ast*.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, `ast.stmt` or `ast.expr`). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, `ast.BinOp` inherits from `ast.expr`. For production rules with alternatives (aka "sums"), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

_fields

Each concrete class has an attribute *_fields* which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, `ast.BinOp` instances have an attribute `left` of type `ast.expr`.

If these attributes are marked as optional in the grammar (using a question mark), the value might be `None`. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with *compile()*.

lineno
col_offset

Instances of `ast.expr` and `ast.stmt` subclasses have *lineno* and *col_offset* attributes. The *lineno* is the line number of source text (1-indexed so the first line is line 1) and the *col_offset* is the UTF-8 byte offset of the first token that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

The constructor of a class `ast.T` parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in `T._fields`; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an `ast.UnaryOp` node, you could use

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

or the more compact

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

33.2.2 Abstract Grammar

The abstract grammar is currently defined as follows:

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        | ClassDef(identifier name,
           expr* bases,
           keyword* keywords,
           stmt* body,
           expr* decorator_list)
        | Return(expr? value)

        | Delete(expr* targets)
        | Assign(expr* targets, 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)
        | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
        | While(expr test, stmt* body, stmt* orelse)
        | If(expr test, stmt* body, stmt* orelse)
        | With(withitem* items, stmt* body)
        | AsyncWith(withitem* items, stmt* body)

        | Raise(expr? exc, expr? cause)
        | Try(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

        -- XXX Jython will be different
        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset)

        -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
          | Compare(expr left, cmpop* ops, expr* comparators)
          | Call(expr func, expr* args, keyword* keywords)
          | Num(object n) -- a number as a PyObject.
          | Str(string s) -- need to specify raw, unicode, etc?
          | FormattedValue(expr value, int? conversion, expr? format_spec)
          | JoinedStr(expr* values)
          | Bytes(bytes s)
          | NameConstant(singleton value)
          | Ellipsis
          | Constant(constant value)

          -- the following expression can appear in assignment context
          | Attribute(expr value, identifier attr, expr_context ctx)
          | Subscript(expr value, slice 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)

           -- col_offset is the byte offset in the utf8 string the parser uses
           attributes (int lineno, int col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    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)

    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)
}
```

33.2.3 ast Helpers

Apart from the node classes, the *ast* module defines these utility functions and classes for traversing abstract syntax trees:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*)
Parse the source into an AST node. Equivalent to `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

> **경고:** It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python's AST compiler.

ast.**literal_eval**(*node_or_string*)
Safely evaluate an expression node or a string containing 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, and `None`.

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

> **경고:** It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python's AST compiler.

버전 3.2에서 변경: Now allows bytes and set literals.

ast.**get_docstring**(*node*, *clean=True*)
Return the docstring of the given *node* (which must be a `FunctionDef`, `AsyncFunctionDef`, `ClassDef`, or `Module` node), or `None` if it has no docstring. If *clean* is true, clean up the docstring's indentation with *inspect.cleandoc()*.

버전 3.5에서 변경: `AsyncFunctionDef` is now supported.

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*)
Increment the line number of each node in the tree starting at *node* by *n*. This is useful to "move code" to a different location in a file.

ast.**copy_location**(*new_node*, *old_node*)
Copy source location (`lineno` and `col_offset`) from *old_node* to *new_node* if possible, and return *new_node*.

ast.**iter_fields**(*node*)
Yield a tuple of `(fieldname, value)` for each field in `node._fields` that is present on *node*.

ast.**iter_child_nodes**(*node*)
Yield all direct child nodes of *node*, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.**walk**(*node*)
Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class ast.**NodeVisitor**
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the *visit()* method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(*node*)
Visit a node. The default implementation calls the method called self.visit_*classname* where *classname* is the name of the node class, or *generic_visit()* if that method doesn't exist.

generic_visit(*node*)
This visitor calls *visit()* on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls *generic_visit()* or visits them itself.

Don't use the *NodeVisitor* if you want to apply changes to nodes during traversal. For this a special visitor exists (*NodeTransformer*) that allows modifications.

class ast.**NodeTransformer**
A *NodeVisitor* subclass that walks the abstract syntax tree and allows modification of nodes.

The *NodeTransformer* will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is None, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (foo) to data['foo']:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=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.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

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))
```

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*)
Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. If *annotate_fields* is true (by default), the returned string will show the names and the values for fields. If *annotate_fields* is false, the result string will be more compact by omitting unambiguous field names. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to true.

더 보기:

Green Tree Snakes, an external documentation resource, has good details on working with Python ASTs.

33.3 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.

33.3.1 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.

33.3.2 심볼 테이블 검사하기

class symtable.**SymbolTable**
블록에 대한 이름 공간 테이블. 생성자는 공개되지 않습니다.

get_type()
심볼 테이블의 형을 돌려줍니다. 가능한 값은 `'class'`, `'module'` 및 `'function'`입니다.

get_id()
테이블의 식별자를 돌려줍니다.

get_name()
테이블의 이름을 돌려줍니다. 테이블이 클래스를 위한 것이면 클래스의 이름이고, 테이블이 함수를 위한 것이면 함수의 이름이고, 테이블이 전역이면 `'top'`입니다 (*get_type()*은 `'module'`을 반환합니다).

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.

has_exec()
블록이 `exec`를 사용하면 `True`를 반환합니다.

get_identifiers()
이 테이블의 심볼 이름들의 리스트를 돌려줍니다.

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_frees()
이 함수의 자유 변수 이름을 포함하는 튜플을 반환합니다.

class symtable.**Class**
클래스의 이름 공간. 이 클래스는 *SymbolTable*을 상속합니다.

get_methods()
클래스에서 선언된 메서드 이름을 포함하는 튜플을 반환합니다.

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_global()
심볼이 전역이면 `True`를 반환합니다.

is_declared_global()
심볼이 global 문으로 전역으로 선언되면 `True`를 반환합니다.

is_local()
심볼이 블록의 지역이면 `True`를 반환합니다.

is_free()
심볼이 블록에서 참조되지만 대입되지 않으면 `True`를 반환합니다.

is_assigned()
심볼이 블록에 대입되면 `True`를 반환합니다.

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. 둘 이상의 이름 공간이 연결되면, *ValueError*가 발생합니다.

33.4 symbol — 파이썬 구문 분석 트리에 사용되는 상수

소스 코드: Lib/symbol.py

이 모듈은 구문 분석 트리의 내부 노드의 숫자 값을 나타내는 상수를 제공합니다. 대부분 파이썬 상수와 달리, 소문자 이름을 사용합니다. 언어 문법의 문맥에서 이름의 정의는 파이썬 배포판의 `Grammar/Grammar` 파일을 참조하십시오. 이름이 매핑되는 특정 숫자 값은 파이썬 버전 간에 변경될 수 있습니다.

이 모듈은 또한 하나의 추가 데이터 객체를 제공합니다:

symbol.**sym_name**
이 모듈에 정의된 상수의 숫자 값을 다시 이름 문자열로 매핑하여, 사람이 읽을 수 있는 구문 분석 트리 표현을 생성할 수 있도록 합니다.

33.5 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/Grammar` 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.

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*)
Return `True` for terminal token values.

token.**ISNONTERMINAL**(*x*)
Return `True` for non-terminal token values.

token.**ISEOF**(*x*)
Return `True` if *x* is the marker indicating the end of input.

토큰 상수는 다음과 같습니다:

token.**ENDMARKER**
token.**NAME**
token.**NUMBER**
token.**STRING**
token.**NEWLINE**
token.**INDENT**
token.**DEDENT**
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**
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**
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.**OP**

token.**ERRORTOKEN**
token.**N_TOKENS**
token.**NT_OFFSET**

다음 토큰 유형 값은 C 토크나이저가 사용하지 않지만 *tokenize* 모듈에 필요합니다.

token.**COMMENT**
주석을 나타내는 데 사용되는 토큰 값.

token.**NL**
비종결 줄넘김을 나타내는데 사용되는 토큰 값. *NEWLINE* 토큰은 파이썬 코드의 논리적 줄의 끝을 나타냅니다; NL 토큰은 코드의 논리적 줄이 여러 물리적 줄로 이어질 때 생성됩니다.

token.**ENCODING**
소스 바이트열을 텍스트로 디코딩하는 데 사용되는 인코딩을 나타내는 토큰 값. *tokenize.tokenize()*에 의해 반환되는 첫 번째 토큰은 항상 ENCODING 토큰입니다.

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.

33.6 keyword — 파이썬 키워드 검사

소스 코드: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword.

keyword.**iskeyword**(*s*)
Return True if *s* is a Python keyword.

keyword.**kwlist**
Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

33.7 tokenize — 파이썬 소스를 위한 토크나이저

소스 코드: Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers", including colorizers for on-screen displays.

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.

33.7.1 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 논리적 줄입니다; 이어지는 줄들이 포함됩니다. 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.

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.

  재구성된 스크립트는 단일 문자열로 반환됩니다. 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.

*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.

  버전 3.2에 추가.

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
```

닫히지 않은 작은따옴표로 묶인 문자열은 에러를 발생시키지 않음에 유의하십시오. 그것들은 *ERRORTOKEN*로 토큰화되고, 그 뒤에 내용이 토큰화됩니다.

33.7.2 명령 줄 사용법

버전 3.3에 추가.

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.

33.7.3 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):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            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:

```
$ python -m tokenize -e 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:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
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      ''
```

Example of tokenizing a file programmatically, reading unicode strings instead of bytes with `generate_tokens()`:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

Or reading bytes directly with *`tokenize()`*:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

33.8 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 내

려가면서, 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.

33.9 pyclbr — Python module browser support

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.

This function is the original interface and is only kept for back compatibility. It returns a filtered version of the following.

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.

버전 3.7에 추가: 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.

33.9.1 Function 객체

클래스 Function 인스턴스는 def 문으로 정의된 함수를 설명합니다. 다음과 같은 어트리뷰트를 가지고 있습니다:

Function.**file**
함수가 정의된 파일의 이름.

Function.**module**
설명된 함수를 정의하는 모듈의 이름.

Function.**name**
함수의 이름.

Function.**lineno**
정의가 시작되는 파일의 줄 번호.

Function.**parent**
최상위 함수면, None. 중첩된 함수면, 부모.

버전 3.7에 추가.

Function.**children**
이름을 중첩된 함수와 클래스에 관한 설명자로 매핑하는 딕셔너리.

버전 3.7에 추가.

33.9.2 Class 객체

클래스 Class 인스턴스는 class 문으로 정의된 클래스를 설명합니다. Function과 같은 어트리뷰트에 더해 두 개의 어트리뷰트가 더 있습니다.

Class.**file**
클래스가 정의된 파일의 이름.

Class.**module**
설명된 클래스를 정의하는 모듈의 이름.

Class.**name**
클래스의 이름.

Class.**lineno**
정의가 시작되는 파일의 줄 번호.

Class.**parent**
최상위 클래스면, None. 중첩된 클래스면, 부모.

버전 3.7에 추가.

Class.**children**
이름을 중첩된 함수와 클래스에 관한 설명자로 매핑하는 딕셔너리.

버전 3.7에 추가.

Class.**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.

Class.**methods**
메서드 이름을 줄 번호에 매핑하는 딕셔너리. 최신 children 딕셔너리에서 파생될 수 있지만, 이전 버전과의 호환성을 위해 남아있습니다.

33.10 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*)
바이트 코드로 소스 파일을 컴파일하고 바이트 코드 캐시 파일을 기록합니다. 소스 코드는 *file*이 지정하는 이름의 파일에서 로드됩니다. 바이트 코드는 *cfile*에 기록되며, 기본값은 .pyc로 끝나는 **PEP 3147/PEP 488** 경로입니다. 예를 들어, *file*이 /foo/bar/baz.py이면 *cfile*은 파이썬 3.2의 경우 /foo/bar/__pycache__/baz.cpython-32.pyc로 기본 설정됩니다. *dfile*이 지정되면, *file* 대신 에러 메시지에서 소스 파일 이름으로 사용됩니다. *doraise*가 참이면, *file*을 컴파일하는 동안 에러를 만나면 *PyCompileError*가 발생합니다. *doraise*가 거짓(기본값)이면, sys.stderr에 에러 문자열이 기록되지만, 예외는 발생하지 않습니다. 이 함수는 바이트 컴파일된 파일의 경로, 즉 사용된 *cfile* 값을 반환합니다.

*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 -1은 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.

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.

버전 3.7에 추가.

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.

py_compile.**main**(*args=None*)
여러 소스 파일을 컴파일합니다. *args*(또는 *args*가 None이면 명령 줄)로 이름 붙여진 파일이 컴파일되고 결과 바이트 코드가 일반적인 방식으로 캐시 됩니다. 이 함수는 소스 파일을 찾기 위해 디렉터리 구조를 검색하지 않습니다; 명시적으로 이름이 지정된 파일 만 컴파일합니다. '-'가 args의 유일한 매개 변수면, 파일 목록을 표준 입력에서 가져옵니다.

버전 3.2에서 변경: '-'에 대한 지원이 추가되었습니다.

이 모듈을 스크립트로 실행하면, *main()*이 명령 줄로 이름이 지정된 모든 파일을 컴파일하는 데 사용됩니다. 파일 중 하나를 컴파일할 수 없으면 종료 상태는 0이 아닙니다.

더 보기:

모듈 ***compileall*** 디렉터리 트리에 있는 모든 파이썬 소스 파일을 컴파일하는 유틸리티.

33.11 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.

33.11.1 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.

-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.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 내용은 pyc-invalidation를 참조하십시오. 기본값은 `SOURCE_DATE_EPOCH` 환경 변수가 설정되지 않으면 `timestamp`이고, `SOURCE_DATE_EPOCH` 환경 변수가 설정되면 `checked-hash`입니다.

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. `-b`는 항상 `.pyc`로 끝나는 바이트 코드 파일을 생성하며, 결코 `.pyo`를 생성하지 않습니다.

버전 3.7에서 변경: `--invalidation-mode` 옵션이 추가되었습니다.

compile() 함수가 사용하는 최적화 수준을 제어하는 명령 줄 옵션은 없습니다. 파이썬 인터프리터 자신이 그 옵션을 제공하고 있기 때문입니다: **python -O -m compileall**.

33.11.2 공용 함수

compileall.**compile_dir**(*dir*, *maxlevels=10*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)

*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 `10`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.

*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.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is lower than `0`, a *ValueError* will be raised.

*invalidation_mode*는 *py_compile.PycInvalidationMode* 열거형의 멤버여야 하며 실행 시간에 생성된 pyc가 무효가 되는 방식을 제어합니다.

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.

compileall.**compile_file**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)

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.

*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.

*invalidation_mode*는 `py_compile.PycInvalidationMode` 열거형의 멤버여야 하며 실행 시간에 생성된 pyc가 무효가 되는 방식을 제어합니다.

버전 3.2에 추가.

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.

`compileall.`**`compile_path`**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)

`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.

`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)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

더 보기:

모듈 ***py_compile*** 단일 소스 파일을 바이트 컴파일합니다.

33.12 dis — Disassembler for Python bytecode

Source code: Lib/dis.py

The *dis* module supports the analysis of CPython *bytecode* by disassembling it. The CPython bytecode which this module takes as an input is defined in the file `Include/opcode.h` and used by the compiler and the interpreter.

CPython implementation detail: Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

버전 3.6에서 변경: Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

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           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

(The "2" is a line number).

33.12.1 Bytecode analysis

버전 3.4에 추가.

The bytecode analysis API allows pieces of Python code to be wrapped in a *Bytecode* object that provides easy access to details of the compiled code.

class dis.**Bytecode**(*x*, *, *first_line=None*, *current_offset=None*)

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by *compile()*).

This is a convenience wrapper around many of the functions listed below, most notably *get_instructions()*, as iterating over a *Bytecode* instance yields the bytecode operations as *Instruction* instances.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If *current_offset* is not `None`, it refers to an instruction offset in the disassembled code. Setting this means *dis()* will display a "current instruction" marker against the specified opcode.

classmethod from_traceback(*tb*)

Construct a *Bytecode* instance from the given traceback, setting *current_offset* to the instruction responsible for the exception.

codeobj

The compiled code object.

first_line

The first source line of the code object (if available)

dis()
Return a formatted view of the bytecode operations (the same as printed by *dis.dis()*, but returned as a multi-line string).

info()
Return a formatted multi-line string with detailed information about the code object, like *code_info()*.

버전 3.7에서 변경: This can now handle coroutine and asynchronous generator objects.

Example:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

33.12.2 Analysis functions

The *dis* module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn't useful:

dis.**code_info**(*x*)
Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

버전 3.2에 추가.

버전 3.7에서 변경: This can now handle coroutine and asynchronous generator objects.

dis.**show_code**(*x*, *, *file=None*)
Print detailed code object information for the supplied function, method, source code string or code object to *file* (or `sys.stdout` if *file* is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

버전 3.2에 추가.

버전 3.4에서 변경: Added *file* parameter.

dis.**dis**(*x=None*, *, *file=None*, *depth=None*)
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 (the code of comprehensions, generator expressions and nested functions, and the code used for building nested classes). 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.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

버전 3.4에서 변경: Added *file* parameter.

버전 3.7에서 변경: Implemented recursive disassembling and added *depth* parameter.

버전 3.7에서 변경: This can now handle coroutine and asynchronous generator objects.

dis.**distb**(*tb=None*, *, *file=None*)
Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

버전 3.4에서 변경: Added *file* parameter.

dis.**disassemble**(*code*, *lasti=-1*, *, *file=None*)
dis.**disco**(*code*, *lasti=-1*, *, *file=None*)
Disassemble a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

버전 3.4에서 변경: Added *file* parameter.

dis.**get_instructions**(*x*, *, *first_line=None*)
Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of *Instruction* named tuples giving the details of each operation in the supplied code.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

버전 3.4에 추가.

dis.**findlinestarts**(*code*)
This generator function uses the `co_firstlineno` and `co_lnotab` attributes 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. See Objects/lnotab_notes.txt for the `co_lnotab` format and how to decode it.

버전 3.6에서 변경: Line numbers can be decreasing. Before, they were always increasing.

dis.**findlabels**(*code*)
Detect all offsets in the raw compiled bytecode string *code* which are jump targets, and return a list of these offsets.

dis.**stack_effect**(*opcode*[, *oparg*])
Compute the stack effect of *opcode* with argument *oparg*.

버전 3.4에 추가.

33.12.3 Python Bytecode Instructions

The *get_instructions()* function and *Bytecode* class provide details of bytecode instructions as *Instruction* instances:

class dis.**Instruction**
Details for a bytecode operation

opcode
numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the *Opcode collections*.

opname
human readable name for operation

arg
numeric argument to operation (if any), otherwise `None`

argval
resolved arg value (if known), otherwise same as arg

argrepr
human readable description of operation argument

offset
start index of operation within bytecode sequence

starts_line
line started by this opcode (if any), otherwise `None`

is_jump_target
`True` if other code jumps to here, otherwise `False`

버전 3.4에 추가.

The Python compiler currently generates the following bytecode instructions.

General instructions

NOP
Do nothing code. Used as a placeholder by the bytecode optimizer.

POP_TOP
Removes the top-of-stack (TOS) item.

ROT_TWO
Swaps the two top-most stack items.

ROT_THREE
Lifts second and third stack item one position up, moves top down to position three.

DUP_TOP
Duplicates the reference on top of the stack.

버전 3.2에 추가.

DUP_TOP_TWO
Duplicates the two references on top of the stack, leaving them in the same order.

버전 3.2에 추가.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE
Implements `TOS = +TOS`.

UNARY_NEGATIVE
Implements `TOS = -TOS`.

UNARY_NOT
Implements `TOS = not TOS`.

UNARY_INVERT
Implements `TOS = ~TOS`.

GET_ITER
Implements `TOS = iter(TOS)`.

GET_YIELD_FROM_ITER
If `TOS` is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements `TOS = iter(TOS)`.

버전 3.5에 추가.

Binary operations

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER
Implements `TOS = TOS1 ** TOS`.

BINARY_MULTIPLY
Implements `TOS = TOS1 * TOS`.

BINARY_MATRIX_MULTIPLY
Implements `TOS = TOS1 @ TOS`.

버전 3.5에 추가.

BINARY_FLOOR_DIVIDE
Implements `TOS = TOS1 // TOS`.

BINARY_TRUE_DIVIDE
Implements `TOS = TOS1 / TOS`.

BINARY_MODULO
Implements `TOS = TOS1 % TOS`.

BINARY_ADD
Implements `TOS = TOS1 + TOS`.

BINARY_SUBTRACT
Implements `TOS = TOS1 - TOS`.

BINARY_SUBSCR
Implements `TOS = TOS1[TOS]`.

BINARY_LSHIFT
Implements `TOS = TOS1 << TOS`.

BINARY_RSHIFT
Implements `TOS = TOS1 >> TOS`.

BINARY_AND
Implements `TOS = TOS1 & TOS`.

BINARY_XOR
Implements `TOS = TOS1 ^ TOS`.

BINARY_OR
Implements `TOS = TOS1 | TOS`.

In-place operations

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

INPLACE_POWER
Implements in-place `TOS = TOS1 ** TOS`.

INPLACE_MULTIPLY
Implements in-place `TOS = TOS1 * TOS`.

INPLACE_MATRIX_MULTIPLY
Implements in-place `TOS = TOS1 @ TOS`.

버전 3.5에 추가.

INPLACE_FLOOR_DIVIDE
Implements in-place `TOS = TOS1 // TOS`.

INPLACE_TRUE_DIVIDE
Implements in-place `TOS = TOS1 / TOS`.

INPLACE_MODULO
Implements in-place `TOS = TOS1 % TOS`.

INPLACE_ADD
Implements in-place `TOS = TOS1 + TOS`.

INPLACE_SUBTRACT
Implements in-place `TOS = TOS1 - TOS`.

INPLACE_LSHIFT
Implements in-place `TOS = TOS1 << TOS`.

INPLACE_RSHIFT
Implements in-place `TOS = TOS1 >> TOS`.

INPLACE_AND
Implements in-place `TOS = TOS1 & TOS`.

INPLACE_XOR
Implements in-place `TOS = TOS1 ^ TOS`.

INPLACE_OR
Implements in-place `TOS = TOS1 | TOS`.

STORE_SUBSCR
Implements `TOS1[TOS] = TOS2`.

DELETE_SUBSCR
Implements `del TOS1[TOS]`.

Coroutine opcodes

GET_AWAITABLE
Implements `TOS = get_awaitable(TOS)`, 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__`.

버전 3.5에 추가.

GET_AITER
Implements `TOS = TOS.__aiter__()`.

버전 3.5에 추가.

버전 3.7에서 변경: Returning awaitable objects from `__aiter__` is no longer supported.

GET_ANEXT
Implements `PUSH(get_awaitable(TOS.__anext__()))`. See `GET_AWAITABLE` for details about `get_awaitable`

버전 3.5에 추가.

BEFORE_ASYNC_WITH
Resolves `__aenter__` and `__aexit__` from the object on top of the stack. Pushes `__aexit__` and result of `__aenter__()` to the stack.

버전 3.5에 추가.

SETUP_ASYNC_WITH
Creates a new frame object.

버전 3.5에 추가.

Miscellaneous opcodes

PRINT_EXPR
Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with *POP_TOP*.

BREAK_LOOP
Terminates a loop due to a `break` statement.

CONTINUE_LOOP (*target*)
Continues a loop due to a `continue` statement. *target* is the address to jump to (which should be a *FOR_ITER* instruction).

SET_ADD (*i*)
Calls `set.add(TOS1[-i], TOS)`. Used to implement set comprehensions.

LIST_APPEND (*i*)
Calls `list.append(TOS[-i], TOS)`. Used to implement list comprehensions.

MAP_ADD (*i*)
Calls `dict.setitem(TOS1[-i], TOS, TOS1)`. Used to implement dict comprehensions.

버전 3.1에 추가.

For all of the *SET_ADD*, *LIST_APPEND* and *MAP_ADD* instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE
Returns with TOS to the caller of the function.

YIELD_VALUE
Pops TOS and yields it from a *generator*.

YIELD_FROM
Pops TOS and delegates to it as a subiterator from a *generator*.

버전 3.3에 추가.

SETUP_ANNOTATIONS
Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains *variable annotations* statically.

버전 3.6에 추가.

IMPORT_STAR
Loads all symbols not starting with `'_'` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements `from module import *`.

POP_BLOCK
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

POP_EXCEPT
Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

END_FINALLY
Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

LOAD_BUILD_CLASS
Pushes `builtins.__build_class__()` onto the stack. It is later called by *CALL_FUNCTION* to construct a class.

SETUP_WITH (*delta*)
This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by `WITH_CLEANUP`. Then, `__enter__()` is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (*POP_TOP*), or store it in (a) variable(s) (*STORE_FAST*, *STORE_NAME*, or *UNPACK_SEQUENCE*).

버전 3.2에 추가.

WITH_CLEANUP_START
Cleans up the stack when a `with` statement block exits. TOS is the context manager's `__exit__()` bound method. Below TOS are 1–3 values indicating how/why the finally clause was entered:

- SECOND = `None`
- (SECOND, THIRD) = (`WHY_{RETURN,CONTINUE}`), retval
- SECOND = `WHY_*`; no retval below it
- (SECOND, THIRD, FOURTH) = exc_info()

In the last case, `TOS(SECOND, THIRD, FOURTH)` is called, otherwise `TOS(None, None, None)`. Pushes SECOND and result of the call to the stack.

WITH_CLEANUP_FINISH
Pops exception type and result of 'exit' function call from the stack.

If the stack represents an exception, *and* the function call returns a 'true' value, this information is "zapped" and replaced with a single `WHY_SILENCED` to prevent *END_FINALLY* from re-raising the exception. (But non-local gotos will still be resumed.)

All of the following opcodes use their arguments.

STORE_NAME (*namei*)
Implements `name = TOS`. *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 TOS into *count* individual values, which are put onto the stack right-to-left.

UNPACK_EX (*counts*)
Implements assignment with a starred target: Unpacks an iterable in TOS 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 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 resulting values are put onto the stack right-to-left.

STORE_ATTR (*namei*)
Implements `TOS.name = TOS1`, where *namei* is the index of name in `co_names`.

DELETE_ATTR (*namei*)
Implements `del TOS.name`, using *namei* as index into `co_names`.

STORE_GLOBAL (*namei*)
Works as *STORE_NAME*, but stores the name as a global.

DELETE_GLOBAL (*namei*)
Works as *DELETE_NAME*, but deletes a global name.

LOAD_CONST (*consti*)
Pushes `co_consts[consti]` onto the stack.

LOAD_NAME (*namei*)
Pushes the value associated with `co_names[namei]` onto the stack.

BUILD_TUPLE (*count*)
Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack.

BUILD_LIST (*count*)
Works as *BUILD_TUPLE*, but creates a list.

BUILD_SET (*count*)
Works as *BUILD_TUPLE*, but creates a set.

BUILD_MAP (*count*)
Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., TOS3: TOS2, TOS1: TOS}`.

버전 3.5에서 변경: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_CONST_KEY_MAP (*count*)
The version of *BUILD_MAP* specialized for constant keys. Pops the top element on the stack which contains a tuple of keys, then starting from `TOS1`, pops *count* values to form values in the built dictionary.

버전 3.6에 추가.

BUILD_STRING (*count*)
Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

버전 3.6에 추가.

BUILD_TUPLE_UNPACK (*count*)
Pops *count* iterables from the stack, joins them in a single tuple, and pushes the result. Implements iterable unpacking in tuple displays `(*x, *y, *z)`.

버전 3.5에 추가.

BUILD_TUPLE_UNPACK_WITH_CALL (*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but is used for `f(*x, *y, *z)` call syntax. The stack item at position `count + 1` should be the corresponding callable `f`.

버전 3.6에 추가.

BUILD_LIST_UNPACK (*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a list instead of tuple. Implements iterable unpacking in list displays `[*x, *y, *z]`.

버전 3.5에 추가.

BUILD_SET_UNPACK (*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a set instead of tuple. Implements iterable unpacking in set displays `{*x, *y, *z}`.

버전 3.5에 추가.

BUILD_MAP_UNPACK (*count*)
Pops *count* mappings from the stack, merges them into a single dictionary, and pushes the result. Implements dictionary unpacking in dictionary displays `{**x, **y, **z}`.

버전 3.5에 추가.

BUILD_MAP_UNPACK_WITH_CALL (*count*)
This is similar to *BUILD_MAP_UNPACK*, but is used for `f(**x, **y, **z)` call syntax. The stack item at position `count + 2` should be the corresponding callable `f`.

버전 3.5에 추가.

버전 3.6에서 변경: The position of the callable is determined by adding 2 to the opcode argument instead of encoding it in the second byte of the argument.

LOAD_ATTR (*namei*)
Replaces TOS with `getattr(TOS, co_names[namei])`.

COMPARE_OP (*opname*)
Performs a Boolean operation. The operation name can be found in `cmp_op[opname]`.

IMPORT_NAME (*namei*)
Imports the module `co_names[namei]`. TOS and TOS1 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 TOS. The resulting object is pushed onto the stack, to be subsequently stored by a *STORE_FAST* instruction.

JUMP_FORWARD (*delta*)
Increments bytecode counter by *delta*.

POP_JUMP_IF_TRUE (*target*)
If TOS is true, sets the bytecode counter to *target*. TOS is popped.

버전 3.1에 추가.

POP_JUMP_IF_FALSE (*target*)
If TOS is false, sets the bytecode counter to *target*. TOS is popped.

버전 3.1에 추가.

JUMP_IF_TRUE_OR_POP (*target*)
If TOS is true, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

버전 3.1에 추가.

JUMP_IF_FALSE_OR_POP (*target*)
If TOS is false, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

버전 3.1에 추가.

JUMP_ABSOLUTE (*target*)
Set bytecode counter to *target*.

FOR_ITER (*delta*)
TOS 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 TOS is popped, and the byte code counter is incremented by *delta*.

LOAD_GLOBAL (*namei*)
Loads the global named `co_names[namei]` onto the stack.

SETUP_LOOP (*delta*)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of *delta* bytes.

SETUP_EXCEPT (*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the first except block.

SETUP_FINALLY (*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the finally block.

LOAD_FAST (*var_num*)
Pushes a reference to the local `co_varnames[var_num]` onto the stack.

STORE_FAST (*var_num*)
Stores TOS into the local `co_varnames[var_num]`.

DELETE_FAST (*var_num*)
Deletes local `co_varnames[var_num]`.

LOAD_CLOSURE (*i*)
Pushes a reference to the cell contained in slot *i* of the cell and free variable storage. The name of the variable is `co_cellvars[i]` if *i* is less than the length of *co_cellvars*. Otherwise it is `co_freevars[i - len(co_cellvars)]`.

LOAD_DEREF (*i*)
Loads the cell contained in slot *i* of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

LOAD_CLASSDEREF (*i*)
Much like *LOAD_DEREF* but first checks the locals dictionary before consulting the cell. This is used for loading free variables in class bodies.

버전 3.4에 추가.

STORE_DEREF (*i*)
Stores TOS into the cell contained in slot *i* of the cell and free variable storage.

DELETE_DEREF (*i*)
Empties the cell contained in slot *i* of the cell and free variable storage. Used by the `del` statement.

버전 3.2에 추가.

RAISE_VARARGS (*argc*)
Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc*:

- 0: `raise` (re-raise previous exception)
- 1: `raise TOS` (raise exception instance or type at `TOS`)
- 2: `raise TOS1 from TOS` (raise exception instance or type at `TOS1` with `__cause__` set to `TOS`)

CALL_FUNCTION (*argc*)
Calls a callable object with positional arguments. *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable object to call. `CALL_FUNCTION` 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.

버전 3.6에서 변경: This opcode is used only for calls with positional arguments.

CALL_FUNCTION_KW (*argc*)
Calls a callable object with positional (if any) and keyword arguments. *argc* indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple of keyword argument names. Below that are keyword arguments in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. `CALL_FUNCTION_KW` 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.

버전 3.6에서 변경: Keyword arguments are packed in a tuple instead of a dictionary, *argc* indicates the total number of arguments.

CALL_FUNCTION_EX (*flags*)
Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Below that is an iterable object containing positional arguments and a callable object to call. *BUILD_MAP_UNPACK_WITH_CALL* and *BUILD_TUPLE_UNPACK_WITH_CALL* can be used for merging multiple mapping objects and iterables containing arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` 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.

버전 3.6에 추가.

LOAD_METHOD (*namei*)
Loads a method named `co_names[namei]` from the TOS object. TOS is popped. This bytecode distinguishes two cases: if TOS has a method with the correct name, the bytecode pushes the unbound method and TOS. TOS will be used as the first argument (`self`) by *CALL_METHOD* when calling the unbound method. Otherwise, `NULL` and the object return by the attribute lookup are pushed.

버전 3.7에 추가.

CALL_METHOD (*argc*)
Calls a method. *argc* is the number of positional arguments. Keyword arguments are not supported. This opcode is designed to be used with *LOAD_METHOD*. Positional arguments are on top of the stack. Below them, the two items described in *LOAD_METHOD* are on the stack (either `self` and an unbound method object or `NULL` and an arbitrary callable). All of them are popped and the return value is pushed.

버전 3.7에 추가.

MAKE_FUNCTION (*flags*)
Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters' default values
- `0x04` an annotation dictionary

- 0x08 a tuple containing cells for free variables, making a closure
- the code associated with the function (at TOS1)
- the *qualified name* of the function (at TOS)

BUILD_SLICE (*argc*)
Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, slice(TOS1, TOS) is pushed; if it is 3, slice(TOS2, TOS1, TOS) is pushed. See the *slice()* built-in function for more information.

EXTENDED_ARG (*ext*)
Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal EXTENDED_ARG are allowed, forming an argument from two-byte to four-byte.

FORMAT_VALUE (*flags*)
Used for implementing formatted literal strings (f-strings). Pops an optional *fmt_spec* from the stack, then a required *value*. *flags* is interpreted as follows:

- (flags & 0x03) == 0x00: *value* is formatted as-is.
- (flags & 0x03) == 0x01: call *str()* on *value* before formatting it.
- (flags & 0x03) == 0x02: call *repr()* on *value* before formatting it.
- (flags & 0x03) == 0x03: call *ascii()* on *value* before formatting it.
- (flags & 0x04) == 0x04: pop *fmt_spec* from the stack and use it, else use an empty *fmt_spec*.

Formatting is performed using PyObject_Format(). The result is pushed on the stack.

버전 3.6에 추가.

HAVE_ARGUMENT
This is not really an opcode. It identifies the dividing line between opcodes which don't use their argument and those that do (< HAVE_ARGUMENT and >= HAVE_ARGUMENT, respectively).

버전 3.6에서 변경: Now every instruction has an argument, but opcodes < HAVE_ARGUMENT ignore it. Before, only opcodes >= HAVE_ARGUMENT had an argument.

33.12.4 Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

dis.**opname**
Sequence of operation names, indexable using the bytecode.

dis.**opmap**
Dictionary mapping operation names to bytecodes.

dis.**cmp_op**
Sequence of all compare operation names.

dis.**hasconst**
Sequence of bytecodes that access a constant.

dis.**hasfree**
Sequence of bytecodes that access a free variable (note that '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**
Sequence of bytecodes that access an attribute by name.

dis.**hasjrel**
Sequence of bytecodes that have a relative jump target.

dis.**hasjabs**
Sequence of bytecodes that have an absolute jump target.

dis.**haslocal**
Sequence of bytecodes that access a local variable.

dis.**hascompare**
Sequence of bytecodes of Boolean operations.

33.13 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.

33.13.1 명령 줄 사용법

버전 3.2에 추가.

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
```

명령 줄 옵션

-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>
하나 이상의 피클 파일이 지정될 때, 각 역 어셈블리 전에 주어진 프리앰블을 인쇄합니다.

33.13.2 프로그래밍 인터페이스

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 3.2에 추가: *annotate* 인자.

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 34

기타 서비스

이 장에서 설명하는 모듈은 모든 파이썬 버전에서 사용할 수 있는 기타 서비스를 제공합니다. 다음은 개요입니다:

34.1 `formatter` — Generic output formatting

버전 3.4부터 폐지: Due to lack of usage, the formatter module has been deprecated.

This module supports two interface definitions, each with multiple implementations: The *formatter* interface, and the *writer* interface which is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of "change back" operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

34.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

formatter.**AS_IS**
Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the `AS_IS` value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

formatter.**writer**
The writer instance with which the formatter interacts.

formatter.**end_paragraph**(*blanklines*)
Close any open paragraphs and insert at least *blanklines* before the next paragraph.

formatter.**add_line_break**()
Add a hard line break if one does not already exist. This does not break the logical paragraph.

formatter.**add_hor_rule**(**args*, ***kw*)
Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

formatter.**add_flowing_data**(*data*)
Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to *add_flowing_data()* is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.**add_literal_data**(*data*)
Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of *data*.

formatter.**add_label_data**(*format*, *counter*)
Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the *format* value is a string, it is interpreted as a format specification for *counter*, which should be an integer. The result of this formatting becomes the value of the label; if *format* is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's `send_label_data()` method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `'1'` represents the counter value formatter as an Arabic number, the characters `'A'` and `'a'` represent alphabetic representations of the counter value in upper and lower case, respectively, and `'I'` and `'i'` represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

formatter.**flush_softspace**()
Send any pending whitespace buffered from a previous call to *add_flowing_data()* to the associated writer object. This should be called before any direct manipulation of the writer object.

formatter.**push_alignment**(*align*)
Push a new alignment setting onto the alignment stack. This may be *AS_IS* if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the *align* value.

formatter.**pop_alignment**()
Restore the previous alignment.

formatter.**push_font**(*(size*, *italic*, *bold*, *teletype)*)
Change some or all font properties of the writer object. Properties which are not set to *AS_IS* are set to the values passed in while others are maintained at their current settings. The writer's new_font() method is called with the fully resolved font specification.

formatter.**pop_font**()
Restore the previous font.

formatter.**push_margin**(*margin*)
Increase the number of left margin indentations by one, associating the logical tag *margin* with the new indentation. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than *AS_IS* are not sufficient to change the margin.

formatter.**pop_margin**()
Restore the previous margin.

formatter.**push_style**(**styles*)
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including *AS_IS* values, is passed to the writer's new_styles() method.

formatter.**pop_style**(*n=1*)
Pop the last *n* style specifications passed to *push_style()*. A tuple representing the revised stack, including *AS_IS* values, is passed to the writer's new_styles() method.

formatter.**set_spacing**(*spacing*)
Set the spacing style for the writer.

formatter.**assert_line_data**(*flag=1*)
Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional *flag* argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

34.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class formatter.**NullFormatter**(*writer=None*)
A formatter which does nothing. If *writer* is omitted, a *NullWriter* instance is created. No methods of the writer are called by *NullFormatter* instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

class formatter.**AbstractFormatter**(*writer*)
The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

34.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the *AbstractFormatter* class as a formatter, the writer must typically be provided by the application.

writer.**flush**()
Flush any buffered output or device control events.

writer.**new_alignment**(*align*)
Set the alignment style. The *align* value can be any object, but by convention is a string or None, where None indicates that the writer's "preferred" alignment should be used. Conventional *align* values are 'left', 'center', 'right', and 'justify'.

writer.**new_font**(*font*)
Set the font style. The value of *font* will be None, indicating that the device's default font should be used, or a tuple of the form (size, italic, bold, teletype). Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are Boolean values specifying which of those font attributes should be used.

writer.**new_margin**(*margin*, *level*)
Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

writer.**new_spacing**(*spacing*)
Set the spacing style to *spacing*.

writer.**new_styles**(*styles*)
Set additional styles. The *styles* value is a tuple of arbitrary values; the value *AS_IS* should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.**send_line_break**()
Break the current line.

writer.**send_paragraph**(*blankline*)
Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to *send_line_break()* before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

writer.**send_hor_rule**(**args*, ***kw*)
Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via *send_line_break()*.

writer.**send_flowing_data**(*data*)
Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

writer.**send_literal_data**(*data*)
Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the send_formatted_data() interface.

writer.**send_label_data**(*data*)
Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string

values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

34.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the `NullWriter` class.

class formatter.**NullWriter**
A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.**AbstractWriter**
A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.**DumbWriter**(*file=None*, *maxcol=72*)
Simple writer class which writes output on the *file object* passed in as *file* or, if *file* is omitted, on standard output. The output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.

CHAPTER 35

MS 윈도우 특정 서비스

이 장에서는 MS 윈도우 플랫폼에서만 사용 가능한 모듈에 관해 설명합니다.

35.1 `msilib` — Read and write Microsoft Installer files

Source code: Lib/msilib/__init__.py

The *msilib* supports the creation of Microsoft Installer (`.msi`) files. Because these files often contain an embedded "cabinet" file (`.cab`), it also exposes an API to create CAB files. Support for reading `.cab` files is currently not implemented; read support for the `.msi` database is possible.

This package aims to provide complete access to all tables in an `.msi` file, therefore, it is a fairly low-level API. Two primary applications of this package are the *distutils* command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

`msilib.`**`FCICreate`**(*cabname*, *files*)
Create a new CAB file named *cabname*. *files* must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

`msilib.`**`UuidCreate`**()
Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

`msilib.`**`OpenDatabase`**(*path*, *persist*)
Return a new database object by calling MsiOpenDatabase. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`,

MSIDBOPEN_DIRECT, MSIDBOPEN_READONLY, or MSIDBOPEN_TRANSACT, and may include the flag MSIDBOPEN_PATCHFILE. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

msilib.**CreateRecord**(*count*)
Return a new record object by calling MSICreateRecord(). *count* is the number of fields of the record.

msilib.**init_database**(*name*, *schema*, *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer*)
Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

schema must be a module object containing tables and _Validation_records attributes; typically, *msilib.schema* should be used.

The database will contain just the schema and the validation records when this function returns.

msilib.**add_data**(*database*, *table*, *records*)
Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be ints, strings, or instances of the Binary class.

class msilib.**Binary**(*filename*)
Represents entries in the Binary table; inserting such an object using *add_data()* reads the file named *filename* into the table.

msilib.**add_tables**(*database*, *module*)
Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

msilib.**add_stream**(*database*, *name*, *path*)
Add the file *path* into the _Stream table of *database*, with the stream name *name*.

msilib.**gen_uuid**()
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in uppercase).

더 보기:

FCICreate UuidCreate UuidToString

35.1.1 Database Objects

Database.**OpenView**(*sql*)
Return a view object, by calling MSIDatabaseOpenView(). *sql* is the SQL statement to execute.

Database.**Commit**()
Commit the changes pending in the current transaction, by calling MSIDatabaseCommit().

Database.**GetSummaryInformation**(*count*)
Return a new summary information object, by calling MsiGetSummaryInformation(). *count* is the maximum number of updated values.

Database.**Close**()
Close the database object, through MsiCloseHandle().

버전 3.7에 추가.

더 보기:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation MsiCloseHandle

35.1.2 View Objects

View.**Execute**(*params*)
Execute the SQL query of the view, through MSIViewExecute(). If *params* is not None, it is a record describing actual values of the parameter tokens in the query.

View.**GetColumnInfo**(*kind*)
Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). *kind* can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View.**Fetch**()
Return a result record of the query, through calling MsiViewFetch().

View.**Modify**(*kind*, *data*)
Modify the view, by calling MsiViewModify(). *kind* can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

data must be a record describing the new data.

View.**Close**()
Close the view, through MsiViewClose().

더 보기:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

35.1.3 Summary Information Objects

SummaryInformation.**GetProperty**(*field*)
Return a property of the summary, through MsiSummaryInfoGetProperty(). *field* is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation.**GetPropertyCount**()
Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation.**SetProperty**(*field*, *value*)
Set a property through MsiSummaryInfoSetProperty(). *field* can have the same values as in *GetProperty()*, *value* is the new value of the property. Possible value types are integer and string.

SummaryInformation.**Persist**()
Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

더 보기:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

35.1.4 Record Objects

`Record.`**`GetFieldCount`**`()`
Return the number of fields of the record, through `MsiRecordGetFieldCount()`.

`Record.`**`GetInteger`**`(`*field*`)`
Return the value of *field* as an integer where possible. *field* must be an integer.

`Record.`**`GetString`**`(`*field*`)`
Return the value of *field* as a string where possible. *field* must be an integer.

`Record.`**`SetString`**`(`*field*, *value*`)`
Set *field* to *value* through `MsiRecordSetString()`. *field* must be an integer; *value* a string.

`Record.`**`SetStream`**`(`*field*, *value*`)`
Set *field* to the contents of the file named *value*, through `MsiRecordSetStream()`. *field* must be an integer; *value* a string.

`Record.`**`SetInteger`**`(`*field*, *value*`)`
Set *field* to *value* through `MsiRecordSetInteger()`. Both *field* and *value* must be an integer.

`Record.`**`ClearData`**`()`
Set all fields of the record to 0, through `MsiRecordClearData()`.

더 보기:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData

35.1.5 Errors

All wrappers around MSI functions raise `MSIError`; the string inside the exception will contain more detail.

35.1.6 CAB Objects

`class` `msilib.`**`CAB`**`(`*name*`)`
The class *CAB* represents a CAB file. During MSI construction, files will be added simultaneously to the `Files` table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

`append``(`*full*, *file*, *logical*`)`
Add the file with the pathname *full* to the CAB file, under the name *logical*. If there is already a file named *logical*, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

`commit``(`*database*`)`
Generate a CAB file, add it as a stream to the MSI file, put it into the `Media` table, and remove the generated file from the disk.

35.1.7 Directory Objects

class msilib.**Directory**(*database*, *cab*, *basedir*, *physical*, *logical*, *default*[, *componentflags*])
Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through *start_component()*, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be None), the path to the physical directory, and a logical directory name. *default* specifies the DefaultDir slot in the directory table. *componentflags* specifies the default flags that new components get.

start_component(*component=None*, *feature=None*, *flags=None*, *keyfile=None*, *uuid=None*)
Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no *feature* is given, the current feature is used. If no *flags* are given, the directory's default flags are used. If no *keyfile* is given, the KeyPath is left null in the Component table.

add_file(*file*, *src=None*, *version=None*, *language=None*)
Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the *src* file is specified, it is interpreted relative to the current directory. Optionally, a *version* and a *language* can be specified for the entry in the File table.

glob(*pattern*, *exclude=None*)
Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

remove_pyc()
Remove .pyc files on uninstall.

더 보기:

Directory Table File Table Component Table FeatureComponents Table

35.1.8 Features

class msilib.**Feature**(*db*, *id*, *title*, *desc*, *display*, *level=1*, *parent=None*, *directory=None*, *attributes=0*)
Add a new record to the Feature table, using the values *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, and *attributes*. The resulting feature object can be passed to the start_component() method of *Directory*.

set_current()
Make this feature the current feature of *msilib*. New components are automatically added to the default feature, unless a feature is explicitly specified.

더 보기:

Feature Table

35.1.9 GUI classes

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

class msilib.**Control**(*dlg*, *name*)
Base class of the dialog controls. *dlg* is the dialog object the control belongs to, and *name* is the control's name.

event(*event*, *argument*, *condition=1*, *ordering=None*)
Make an entry into the `ControlEvent` table for this control.

mapping(*event*, *attribute*)
Make an entry into the `EventMapping` table for this control.

condition(*action*, *condition*)
Make an entry into the `ControlCondition` table for this control.

class msilib.**RadioButtonGroup**(*dlg*, *name*, *property*)
Create a radio button control named *name*. *property* is the installer property that gets set when a radio button is selected.

add(*name*, *x*, *y*, *width*, *height*, *text*, *value=None*)
Add a radio button named *name* to the group, at the coordinates *x*, *y*, *width*, *height*, and with the label *text*. If *value* is `None`, it defaults to *name*.

class msilib.**Dialog**(*db*, *name*, *x*, *y*, *w*, *h*, *attr*, *title*, *first*, *default*, *cancel*)
Return a new *Dialog* object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control(*name*, *type*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *control_next*, *help*)
Return a new *Control* object. An entry in the `Control` table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

text(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*)
Add and return a `Text` control.

bitmap(*name*, *x*, *y*, *width*, *height*, *text*)
Add and return a `Bitmap` control.

line(*name*, *x*, *y*, *width*, *height*)
Add and return a `Line` control.

pushbutton(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*, *next_control*)
Add and return a `PushButton` control.

radiogroup(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `RadioButtonGroup` control.

checkbox(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `CheckBox` control.

더 보기:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

35.1.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

msilib.**schema**
This is the standard MSI schema for MSI 2.0, with the *tables* variable providing a list of table definitions, and *_Validation_records* providing the data for MSI validation.

msilib.**sequence**
This module contains table contents for the standard sequence tables: *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

msilib.**text**
This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

35.2 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.

35.2.1 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.

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*)
Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of *os.O_APPEND*, *os.O_RDONLY*, and *os.O_TEXT*. The returned file descriptor may be used as a parameter to *os.fdopen()* to create a file object.

msvcrt.**get_osfhandle**(*fd*)
파일 기술자 *fd*의 파일 핸들을 돌려줍니다. *fd*가 인식되지 않으면 *OSError*를 발생시킵니다.

35.2.2 콘솔 I/O

msvcrt.**kbhit**()
Return `True` if a keypress is waiting to be read.

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.

35.2.3 기타 함수

msvcrt.**heapmin**()
강제로 `malloc()` 힙이 자신을 정리하고, 사용하지 않는 블록을 운영 체제로 반환하도록 합니다. 실패하면, *OSError*가 발생합니다.

35.3 winreg — Windows registry access

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a *handle object* is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

버전 3.3에서 변경: Several functions in this module used to raise a `WindowsError`, which is now an alias of `OSError`.

35.3.1 Functions

This module offers the following functions:

winreg.**CloseKey**(*hkey*)
Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

> 참고: If *hkey* is not closed using this method (or via `hkey.Close()`), it is closed when the *hkey* object is destroyed by Python.

winreg.**ConnectRegistry**(*computer_name*, *key*)
Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

버전 3.3에서 변경: See *above*.

winreg.**CreateKey**(*key*, *sub_key*)
Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

버전 3.3에서 변경: See *above*.

winreg.**CreateKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)
Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WRITE`. See *Access Rights* for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

버전 3.2에 추가.

버전 3.3에서 변경: See *above*.

winreg.**DeleteKey**(*key*, *sub_key*)
Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

버전 3.3에서 변경: See *above*.

winreg.**DeleteKeyEx**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)
Deletes the specified key.

참고: The *DeleteKeyEx()* function is implemented with the RegDeleteKeyEx Windows API function, which is specific to 64-bit versions of Windows. See the RegDeleteKeyEx documentation.

---

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_WOW64_64KEY*. See *Access Rights* for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

On unsupported Windows versions, *NotImplementedError* is raised.

버전 3.2에 추가.

버전 3.3에서 변경: See *above*.

winreg.**DeleteValue**(*key*, *value*)
Removes a named value from a registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value is a string that identifies the value to remove.

winreg.**EnumKey**(*key*, *index*)
Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an `OSError` exception is raised, indicating, no more values are available.

버전 3.3에서 변경: See *above*.

winreg.**EnumValue**(*key*, *index*)
Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an `OSError` exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	Meaning
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for `SetValueEx()`)

버전 3.3에서 변경: See *above*.

winreg.**ExpandEnvironmentStrings**(*str*)
Expands environment variable placeholders `%NAME%` in strings like `REG_EXPAND_SZ`:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

winreg.**FlushKey**(*key*)
Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined *HKEY_* constants*.

It is not necessary to call `FlushKey()` to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike `CloseKey()`, the `FlushKey()` method returns only when all the data has been written to the registry. An application should only call `FlushKey()` if it requires absolute certainty that registry changes are on disk.

참고: If you don't know whether a `FlushKey()` call is required, it probably isn't.

winreg.**LoadKey**(*key*, *sub_key*, *file_name*)
Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by `ConnectRegistry()` or one of the constants `HKEY_USERS` or `HKEY_LOCAL_MACHINE`.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to `LoadKey()` fails if the calling process does not have the `SE_RESTORE_PRIVILEGE` privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If *key* is a handle returned by `ConnectRegistry()`, then the path specified in *file_name* is relative to the remote computer.

winreg.**OpenKey**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
winreg.**OpenKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
Opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_READ*. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, *OSError* is raised.

버전 3.2에서 변경: Allow the use of named arguments.

버전 3.3에서 변경: See *above*.

winreg.**QueryInfoKey**(*key*)
Returns information about a key, as a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

The result is a tuple of 3 items:

Index	Meaning
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

winreg.**QueryValue**(*key*, *sub_key*)
Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the *SetValue()* method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a `NULL` name. But the underlying API call doesn't return the type, so always use *QueryValueEx()* if possible.

winreg.**QueryValueEx**(*key*, *value_name*)
Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	Meaning
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for *SetValueEx()*)

winreg.**SaveKey**(*key*, *file_name*)
Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined *HKEY_* constants*.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the `LoadKey()` method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the `SeBackupPrivilege` security privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes `NULL` for *security_attributes* to the API.

winreg.**SetValue**(*key*, *sub_key*, *type*, *value*)
Associates a value with a specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be `REG_SZ`, meaning only strings are supported. Use the `SetValueEx()` function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.

winreg.**SetValueEx**(*key*, *value_name*, *reserved*, *type*, *value*)
Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See *Value Types* for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with `KEY_SET_VALUE` access.

To open the key, use the `CreateKey()` or `OpenKey()` methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

winreg.**DisableReflectionKey**(*key*)
Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

winreg.**EnableReflectionKey**(*key*)
Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

`winreg.`**`QueryReflectionKey`**(*key*)
Determines the reflection state for the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Returns `True` if reflection is disabled.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

35.3.2 Constants

The following constants are defined for use in many `_winreg` functions.

HKEY_* Constants

`winreg.`**`HKEY_CLASSES_ROOT`**
Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

`winreg.`**`HKEY_CURRENT_USER`**
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

`winreg.`**`HKEY_LOCAL_MACHINE`**
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

`winreg.`**`HKEY_USERS`**
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

`winreg.`**`HKEY_PERFORMANCE_DATA`**
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

`winreg.`**`HKEY_CURRENT_CONFIG`**
Contains information about the current hardware profile of the local computer system.

`winreg.`**`HKEY_DYN_DATA`**
This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

winreg.**KEY_ALL_ACCESS**
Combines the STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, and *KEY_CREATE_LINK* access rights.

winreg.**KEY_WRITE**
Combines the STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE*, and *KEY_CREATE_SUB_KEY* access rights.

winreg.**KEY_READ**
Combines the STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, and *KEY_NOTIFY* values.

winreg.**KEY_EXECUTE**
Equivalent to *KEY_READ*.

winreg.**KEY_QUERY_VALUE**
Required to query the values of a registry key.

winreg.**KEY_SET_VALUE**
Required to create, delete, or set a registry value.

winreg.**KEY_CREATE_SUB_KEY**
Required to create a subkey of a registry key.

winreg.**KEY_ENUMERATE_SUB_KEYS**
Required to enumerate the subkeys of a registry key.

winreg.**KEY_NOTIFY**
Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.**KEY_CREATE_LINK**
Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

winreg.**KEY_WOW64_64KEY**
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

winreg.**KEY_WOW64_32KEY**
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see Registry Value Types.

winreg.**REG_BINARY**
Binary data in any form.

winreg.**REG_DWORD**
32-bit number.

winreg.**REG_DWORD_LITTLE_ENDIAN**
A 32-bit number in little-endian format. Equivalent to *REG_DWORD*.

winreg.**REG_DWORD_BIG_ENDIAN**
A 32-bit number in big-endian format.

winreg.**REG_EXPAND_SZ**
Null-terminated string containing references to environment variables (`%PATH%`).

winreg.**REG_LINK**
A Unicode symbolic link.

winreg.**REG_MULTI_SZ**
A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.**REG_NONE**
No defined value type.

winreg.**REG_QWORD**
A 64-bit number.

버전 3.6에 추가.

winreg.**REG_QWORD_LITTLE_ENDIAN**
A 64-bit number in little-endian format. Equivalent to *REG_QWORD*.

버전 3.6에 추가.

winreg.**REG_RESOURCE_LIST**
A device-driver resource list.

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**
A hardware setting.

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**
A hardware resource list.

winreg.**REG_SZ**
A null-terminated string.

35.3.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the *Close()* method on the object, or the *CloseKey()* function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` – thus

```
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in *int()* function), in which case the underlying Windows handle value is returned. You can also use the *Detach()* method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.**Close**()
Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.**Detach**()
Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

PyHKEY.**__enter__**()
PyHKEY.**__exit__**(**exc_info*)
The HKEY object implements __enter__() and __exit__() and thus supports the context protocol for the with statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the with block.

35.4 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 None이면, 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_OK입니다. -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
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
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.**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.

CHAPTER 36

유닉스 특정 서비스

이 장에서 설명하는 모듈은 유닉스 운영 체제(혹은 때에 따라 일부 혹은 많은 변종)에 고유한 기능에 대한 인터페이스를 제공합니다. 다음은 개요입니다:

36.1 `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.

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.

36.1.1 대용량 파일 지원

여러 운영 체제(AIX, HP-UX, Irix 및 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.

Large file support is enabled in Python when the size of an `off_t` is larger than a `long` and the `long long` is at least as large as an `off_t`. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
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
```

36.1.2 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 environ을 *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 environ의 대체 구현을 제공합니다. *os.environ*를 갱신하면 이 딕셔너리를 쓸모없게 만드는 것에 유의하십시오. *os* 모듈 버전을 사용하는 것이 *posix* 모듈에 직접 액세스하는 것보다 권장됩니다.

36.2 pwd — 암호 데이터베이스

이 모듈은 유닉스 사용자 계정과 암호 데이터베이스에 대한 액세스를 제공합니다. 모든 유닉스 버전에서 사용할 수 있습니다.

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_passwd는 대개 DES 파생 알고리즘으로 암호화된 암호를 포함합니다 (모듈 *crypt*를 보세요). 그러나 대부분의 현대 유닉스는 소위 섀도 암호 시스템을 사용합니다. 이러한 유

닉스에서 *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. 사용할 수 있다면, *spwd* 모듈을 암호화된 암호에 대한 액세스가 필요한 곳에 사용해야 합니다.

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.

모듈 ***spwd*** 섀도 암호 데이터베이스와의 인터페이스, 이것과 유사합니다.

36.3 spwd — 섀도 암호 데이터베이스

이 모듈은 유닉스 섀도 암호 데이터베이스에 대한 액세스를 제공합니다. 다양한 유닉스 버전에서 사용할 수 있습니다.

섀도 암호 데이터베이스에 액세스할 수 있는 충분한 권한이 있어야 합니다(일반적으로 루트여야 함을 뜻합니다).

섀도 암호 데이터베이스 항목은 튜플류 객체로 보고됩니다. 어트리뷰트는 `spwd` 구조체의 멤버에 해당합니다 (아래의 어트리뷰트 필드, `<shadow.h>` 참조):

인덱스	어트리뷰트	의미
0	`sp_namp`	로그인 이름
1	`sp_pwdp`	암호화된 암호
2	`sp_lstchg`	최종 변경 날짜
3	`sp_min`	변경 간 최소 일수
4	`sp_max`	변경 간 최대 일수
5	`sp_warn`	암호 만료 며칠 전에 사용자에게 경고할지, 날짜 수로 표현
6	`sp_inact`	암호 만료 후 며칠 후에 계정이 비활성화될지, 날짜 수로 표현
7	`sp_expire`	계정 만료일, 1970-01-01 이후 일수로 표현
8	`sp_flag`	예약됨

sp_namp 및 sp_pwdp 항목은 문자열이며, 다른 모든 항목은 정수입니다. 요청된 항목을 찾을 수 없으면 *KeyError*가 발생합니다.

다음 함수가 정의됩니다:

spwd.**getspnam**(*name*)
지정된 사용자 이름에 대한 섀도 암호 데이터베이스 항목을 반환합니다.

버전 3.6에서 변경: 사용자에게 권한이 없으면 *KeyError* 대신 *PermissionError*를 발생시킵니다.

spwd.**getspall**()
사용 가능한 모든 섀도 암호 데이터베이스 항목의 리스트를 임의의 순서로 반환합니다.

더 보기:

모듈 *grp* 그룹 데이터베이스에 대한 인터페이스, 이것과 유사합니다.

모듈 *pwd* 일반적인 암호 데이터베이스와의 인터페이스, 이것과 유사합니다.

36.4 grp — 그룹 데이터베이스

이 모듈은 유닉스 그룹 데이터베이스에 대한 액세스를 제공합니다. 모든 유닉스 버전에서 사용할 수 있습니다.

그룹 데이터베이스 항목은 `group` 구조체(아래의 어트리뷰트 필드, `<pwd.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`**(*gid*)
주어진 숫자 그룹 ID에 대한 그룹 데이터베이스 항목을 반환합니다. 요청된 항목을 찾을 수 없으면 *KeyError*가 발생합니다.

버전 3.6부터 폐지: 파이썬 3.6부터 *getgrgid()*에서 float나 문자열과 같은 정수가 아닌 인자의 지원은 폐지되었습니다.

`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.

모듈 *spwd* 섀도 암호 데이터베이스와의 인터페이스, 이것과 유사합니다.

36.5 crypt — 유닉스 비밀번호 확인 함수

소스 코드: Lib/crypt.py

이 모듈은 수정된 DES 알고리즘을 기반으로 하는 단방향 해시 함수인 *crypt (3)* 루틴에 대한 인터페이스를 구현합니다; 자세한 내용은 유닉스 매뉴얼 페이지를 참조하십시오. 가능한 용도는 실제 암호를 저장하지 않고 암호를 확인하기 위해 해시 된 암호를 저장하거나 사전으로 유닉스 암호를 해독하려고 시도하는 것을 포함합니다.

이 모듈의 동작은 실행 중인 시스템의 *crypt (3)* 루틴의 실제 구현에 따라 달라집니다. 따라서, 현재 구현에서 사용할 수 있는 모든 확장을 이 모듈에서도 사용할 수 있습니다.

36.5.1 해싱 방법

버전 3.3에 추가.

crypt 모듈은 해싱 방법 목록을 정의합니다 (모든 플랫폼에서 모든 방법을 사용할 수 있는 것은 아닙니다):

crypt.**METHOD_SHA512**
: SHA-512 해시 함수를 기반으로 16문자의 솔트 (salt) 와 86문자의 해시를 사용하는 모듈형 암호 형식 (Modular Crypt Format) 방법. 이것은 가장 강력한 방법입니다.

crypt.**METHOD_SHA256**
: SHA-256 해시 함수를 기반으로 16자의 솔트와 43자의 해시를 사용하는 또 다른 모듈형 암호 형식 방법.

crypt.**METHOD_BLOWFISH**
: Blowfish 암호를 기반으로 22문자의 솔트와 31문자의 해시를 사용하는 또 다른 모듈형 암호 형식 방법.

  버전 3.7에 추가.

crypt.**METHOD_MD5**
: MD5 해시 함수를 기반으로 8자의 솔트와 22자의 해시를 사용하는 또 다른 모듈형 암호 형식 방법.

crypt.**METHOD_CRYPT**
: 2문자의 솔트와 13문자의 해시를 사용하는 전통적인 방법. 이것은 가장 약한 방법입니다.

36.5.2 모듈 어트리뷰트

버전 3.3에 추가.

crypt.**methods**
: 사용 가능한 비밀번호 해싱 알고리즘 리스트, `crypt.METHOD_*` 객체들. 이 리스트는 가장 강한 것부터 가장 약한 것 순으로 정렬됩니다.

36.5.3 모듈 함수

crypt 모듈은 다음 함수를 정의합니다:

crypt.**crypt**(*word*, *salt=None*)
: *word*는 대개 프롬프트나 그래픽 인터페이스에서 입력한 사용자의 비밀번호입니다. 선택적 *salt*는 *mksalt()* 에서 반환된 문자열, `crypt.METHOD_*` 값 중 하나(모든 플랫폼에서 모든 것을 사용할 수 있는 것은 아니지만), 또는 이 함수가 반환하는 솔트를 포함한 전체 암호화된 비밀번호입니다. *salt*가 제공되지 않으면, (*methods()* 에서 반환되는) 가장 강력한 방법이 사용됩니다.

비밀번호 확인은 일반적으로 *word*로 평문 비밀번호를 전달하는 것으로 수행됩니다. 이전 *crypt()* 호출의 전체 결과와 이 호출의 결과가 같아야 합니다.

salt(무작위의 2자나 16자 문자열, 방법을 가리키기 위해 앞에 `$digit$`가 붙을 수 있음)는 암호화 알고리즘을 교란하는 데 사용됩니다. *salt*의 문자는 `$digit$`를 앞에 붙이는 모듈형 암호 형식을 제외하고는 `[./a-zA-Z0-9]` 집합에 있어야 합니다.

해시 된 비밀번호를 문자열로 반환합니다. 이 문자열은 솔트와 같은 알파벳의 문자로 구성됩니다.

몇 가지 *crypt(3)* 확장이 *salt*에 다른 크기의 다른 값을 허용하기 때문에, 암호를 확인할 때 전체 암호화 된 비밀번호를 솔트로 사용하는 것이 좋습니다.

버전 3.3에서 변경: *salt*에 대한 문자열 외에 `crypt.METHOD_*` 값을 받아들입니다.

`crypt.`**`mksalt`**(*method=None*, ***, *rounds=None*)

지정된 방법의 무작위로 생성된 솔트를 반환합니다. *method*가 주어지지 않으면, *methods()*에 의해 반환된 가장 강력한 방법이 사용됩니다.

반환 값은 *salt* 인자로 *crypt()*에 전달하기에 적합한 문자열입니다.

*rounds*는 `METHOD_SHA256`, `METHOD_SHA512` 및 `METHOD_BLOWFISH`에 대한 라운드 수를 지정합니다. `METHOD_SHA256`과 `METHOD_SHA512`의 경우 `1000`와 `999_999_999` 사이의 정수여야 하며, 기본값은 `5000`입니다. `METHOD_BLOWFISH`의 경우 `16`(2^4)과 `2_147_483_648`(2^{31}) 사이의 2의 거듭제곱이어야 하며, 기본값은 `4096`(2^{12}) 입니다.

버전 3.3에 추가.

버전 3.7에서 변경: *rounds* 매개 변수가 추가되었습니다.

36.5.4 예제

일반적인 사용법을 보여주는 간단한 예제 (타이밍 공격에 대한 노출을 제한하기 위해서는 상수 시간 비교 연산이 필요합니다. *hmac.compare_digest()*가 이 목적에 적합합니다):

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    else:
        return True
```

가장 강력한 방법을 사용하여 비밀번호의 해시를 생성하고, 원본과 비교하여 확인하려면 이렇게 합니다:

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

36.6 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.

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: 즉시 변경하려면 `TCSANOW`, 계류 중인 모든 출력을 전송한 후에 변경하려면 `TCSADRAIN`, 계류 중인 모든 출력을 전송하고 계류 중인 모든 입력을 버린 후 변경하려면 `TCSAFLUSH`.

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 `TCION`가 될 수 있습니다.

더 보기:

모듈 *tty* 공통 터미널 제어 연산을 위한 편리 함수.

36.6.1 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
```

36.7 tty — 터미널 제어 함수

소스 코드: Lib/tty.py

---

tty 모듈은 tty를 cbreak 및 raw 모드로 설정하는 함수를 정의합니다.

termios 모듈이 필요하기 때문에 유닉스에서만 작동합니다.

tty 모듈은 다음 함수를 정의합니다.:

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)
파일 기술자 *fd*의 모드를 raw로 변경합니다. *when*이 생략되면, 기본값은 termios.TCSAFLUSH이며 *termios.tcsetattr()*로 전달됩니다.

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)
파일 기술자 *fd*의 모드를 cbreak로 변경합니다. *when*이 생략되면, 기본값은 termios.TCSAFLUSH이며 *termios.tcsetattr()*로 전달됩니다.

더 보기:

모듈 *termios* 저수준 터미널 제어 인터페이스.

36.8 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.

의사 터미널 처리는 플랫폼에 따라 매우 다르므로, 리눅스에서만 수행할 수 있는 코드가 있습니다. (리눅스 코드는 다른 플랫폼에서도 작동하리라고 기대되지만, 아직 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.

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.

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 *OSError*를 발생시켜야 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).

두 콜백이 모두 EOF 신호를 보내면, *select*가 세 개의 빈 리스트를 전달할 때 플랫폼에서 에러를 일으키지 않는 한 *spawn*은 아마도 절대 반환하지 않습니다. 이것은 버그이고, issue 26228에서 설명하고 있습니다.

버전 3.4에서 변경: 이제 *spawn()*은 자식 프로세스에 대한 *os.waitpid()*로부터 온 상태 값을 반환합니다.

36.8.1 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')
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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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)
```

36.9 fcntl — fcntl과 ioctl 시스템 호출

이 모듈은 파일 기술자에 대한 파일 제어와 I/O 제어를 수행합니다. `fcntl()`과 `ioctl()` 유닉스 루틴에 대한 인터페이스입니다. 이 호출에 대한 자세한 설명은 *fcntl(2)*과 *ioctl(2)* 유닉스 매뉴얼 페이지를 참조하십시오.

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.

이 모듈은 다음 함수를 정의합니다:

`fcntl.`**`fcntl`**(*fd*, *cmd*, *arg=0*)

파일 기술자 *fd*(*fileno()* 메서드를 제공하는 파일 객체도 허용됩니다)에 대해 *cmd* 연산을 수행합니다. *cmd*에 사용되는 값은 운영 체제에 따라 다르며, 관련 C 헤더 파일에 사용된 것과 같은 이름을 사용하여 *fcntl* 모듈에서 상수로 제공됩니다. 인자 *arg*는 정숫값이나 *bytes* 객체가 될 수 있습니다. 정숫값일 때, 이 함수의 반환 값은 C `fcntl()` 호출의 정수 반환 값입니다. 인자가 바이트열일 때 바이너리 구조체를 나타냅니다, 예를 들어 *struct.pack()*으로 만든 것입니다. 바이너리 데이터는 주소가 C `fcntl()` 호출에 전달될 버퍼로 복사됩니다. 호출 성공 후 반환 값은 버퍼 내용이며, *bytes* 객체로 변환됩니다. 반환된 객체의 길이는 *arg* 인자의 길이와 같습니다. 이것은 1024바이트로 제한됩니다. 운영 체제에 의해 버퍼로 반환된 정보가 1024바이트보다 크면, 세그멘테이션 위반이나 더 미묘한 데이터 손상이 발생할 가능성이 큽니다.

`fcntl()`이 실패하면, *OSError*가 발생합니다.

`fcntl.`**`ioctl`**(*fd*, *request*, *arg=0*, *mutate_flag=True*)

이 함수는 인자 처리가 훨씬 더 복잡하다는 점을 제외하면, *fcntl()* 함수와 같습니다.

request 매개 변수는 32비트에 맞출 수 있는 값으로 제한됩니다. *request* 인자로 사용하기 위한 추가 상수는 관련 C 헤더 파일에서 사용된 것과 같은 이름으로 *termios* 모듈에서 제공됩니다.

매개 변수 *arg*는 정수, 읽기 전용 버퍼 인터페이스를 지원하는 (*bytes* 같은) 객체 또는 읽기-쓰기 버퍼 인터페이스를 지원하는 (*bytearray* 같은) 객체 중 하나일 수 있습니다.

마지막 경우를 제외하고는, 동작이 *fcntl()* 함수와 같습니다.

가변 버퍼가 전달되면, 동작은 *mutate_flag* 매개 변수의 값에 의해 결정됩니다.

거짓이면, 버퍼의 가변성은 무시되고 동작은 읽기 전용 버퍼일 때와 같습니다. 단, 위에서 언급한 1024바이트 제한은 피할 수 있습니다 – 최소한 전달한 버퍼가 운영 체제가 원하는 만큼 길면 작동해야 합니다.

*mutate_flag*가 참(기본값)이면, 버퍼가 (결과적으로) 하부 *ioctl()* 시스템 호출로 전달되고, 이 호출의 반환 코드는 호출하는 파이썬으로 다시 전달되고 버퍼의 새로운 내용은 *ioctl()*의 동작을 반영합니다.

이것은 약간 단순화한 설명인데, 제공된 버퍼가 1024바이트보다 작으면, 1024바이트 길이의 정적 버퍼에 먼저 복사된 다음, 이 정적 버퍼가 *ioctl()*로 전달되고, 정적 버퍼를 제공된 버퍼로 다시 복사하기 때문입니다.

ioctl()이 실패하면, *OSError* 예외가 발생합니다.

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])
```

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()이 실패하면, *OSError* 예외가 발생합니다.

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, *cmd*는 다음 값 중 하나입니다:

- LOCK_UN – 잠금 해제
- LOCK_SH – 공유 잠금 획득
- LOCK_EX – 배타적 잠금 획득

*cmd*가 LOCK_SH나 LOCK_EX 일 때, 잠금 획득시 블로킹을 피하고자 LOCK_NB와 비트별 OR 될 수 있습니다. LOCK_NB가 사용되고 잠금을 얻을 수 없을 때, *OSError*가 발생하고 *errno* 어트리뷰트가 EACCES나 EAGAIN으로 설정됩니다(운영 체제에 따라 다릅니다; 이식성을 위해서 두 값을 모두 확인하십시오). 적어도 일부 시스템에서, LOCK_EX는 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.

예제 (모두 SVR4 호환 시스템에서):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.

36.10 pipes — 셸 파이프라인에 대한 인터페이스

소스 코드: Lib/pipes.py

---

pipes 모듈은 파이프라인 개념을 추상화하는 클래스를 정의합니다 — 하나의 파일을 다른 파일로 변환하는 일련의 변환기입니다.

모듈이 **/bin/sh** 명령 줄을 사용하므로, *os.system()* 와 *os.popen()*를 위한 POSIX 나 그와 호환되는 셸이 필요합니다.

pipes 모듈은 다음 클래스를 정의합니다:

class pipes.**Template**
파이프라인의 추상화.

예제:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

36.10.1 Template 객체

Template 객체는 다음 메서드를 갖습니다:

Template.**reset**()
파이프라인 템플릿을 초기 상태로 복원합니다.

Template.**clone**()
새로운 동등한 파이프라인 템플릿을 반환합니다.

Template.**debug**(*flag*)
*flag*가 참이면, 디버깅을 켭니다. 그렇지 않으면, 디버깅을 끕니다. 디버깅이 켜지면, 실행되는 명령이 인쇄되고, 셸에 `set -x` 명령이 주어져 더 자세한 정보가 표시됩니다.

Template.**append**(*cmd*, *kind*)
끝에 새로운 액션을 추가합니다. *cmd* 변수는 올바른 bourne 셸 명령이어야 합니다. *kind* 변수는 두 개의 문자로 구성됩니다.

첫 번째 문자는 `'-'` (명령이 표준 입력을 읽음을 의미), `'f'` (명령이 명령 줄에서 주어진 파일을 읽음을 의미) 또는 `'.'` (명령이 입력을 읽지 않음을 의미하므로, 반드시 첫 번째여야 합니다) 일 수 있습니다.

마찬가지로, 두 번째 문자는 `'-'` (명령이 표준 출력에 쓰는 것을 의미), `'f'` (명령이 명령 줄에서 주어진 파일에 쓰는 것을 의미) 또는 `'.'` (명령이 아무것도 쓰지 않음을 의미하므로, 반드시 마지막이어야 합니다) 일 수 있습니다.

Template.**prepend**(*cmd*, *kind*)
처음에 새로운 액션을 추가합니다. 인자에 대한 설명은 *append()*를 참조하십시오.

Template.**open**(*file*, *mode*)
*file*로 열려 있지만, 파이프라인에서 읽거나 파이프라인으로 쓰는 파일류 객체를 반환합니다. `'r'`, `'w'` 중 하나만 주어질 수 있습니다.

Template.**copy**(*infile*, *outfile*)
파이프를 통해 *infile*를 *outfile*로 복사합니다.

36.11 resource — 자원 사용 정보

이 모듈은 프로그램에서 사용하는 시스템 자원을 측정하고 제어하기 위한 기본 메커니즘을 제공합니다.

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.

36.11.1 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.

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.

가용성: glibc 2.13 이상이 설치된 리눅스 2.6.36 이상.

버전 3.4에 추가.

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.

resource.**RLIMIT_AS**
프로세스에서 사용할 수 있는 주소 공간의 최대 영역 (바이트).

resource.**RLIMIT_MSGQUEUE**
POSIX 메시지 큐에 할당할 수 있는 바이트 수.

가용성: 리눅스 2.6.8 이상.

버전 3.4에 추가.

resource.**RLIMIT_NICE**
프로세스의 나이스(nice) 수준의 상한 (20 - rlim_cur로 계산됩니다).

가용성: 리눅스 2.6.12 이상.

버전 3.4에 추가.

resource.**RLIMIT_RTPRIO**
실시간 우선순위의 상한.

가용성: 리눅스 2.6.12 이상.

버전 3.4에 추가.

resource.**RLIMIT_RTTIME**
프로세스가 블로킹 시스템 호출 없이 실시간 스케줄링 하에서 소비할 수 있는 CPU 시간의 시간제한 (마이크로초).

가용성: 리눅스 2.6.25 이상.

버전 3.4에 추가.

resource.**RLIMIT_SIGPENDING**
프로세스가 큐에 넣을 수 있는 시그널 수입니다.

가용성: 리눅스 2.6.8 이상.

버전 3.4에 추가.

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 9 이상.

버전 3.4에 추가.

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 9 이상.

버전 3.4에 추가.

resource.**RLIMIT_NPTS**
이 사용자 ID로 만들어지는 최대 의사 터미널(pseudo-terminal) 수.

가용성: FreeBSD 9 이상.

버전 3.4에 추가.

36.11.2 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.

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_utime과 ru_stime은 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	time in user mode (float)
1	ru_stime	time in system mode (float)
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.

버전 3.2에 추가.

36.12 nis — Sun의 NIS(옐로 페이지)에 대한 인터페이스

---

nis 모듈은 여러 호스트의 중앙 관리에 유용한 NIS 라이브러리를 감싸는 얇은 래퍼를 제공합니다.

NIS가 유닉스 시스템에만 존재하므로, 이 모듈은 유닉스에서만 사용할 수 있습니다.

nis 모듈은 다음 함수를 정의합니다:

nis.**match**(*key*, *mapname*, *domain=default_domain*)
맵 *mapname*에서 *key*에 대한 일치를 반환하거나, 일치가 없으면 에러(*nis.error*)를 발생시킵니다. 둘 다 문자열이어야 하며, *key*는 8비트 클린해야 합니다. 반환 값은 임의의 바이트 배열입니다 (NULL 이나 다른 기쁨을 포함할 수 있습니다).

*mapname*이 다른 이름의 별칭인지 먼저 검사합니다.

domain 인자는 조회에 사용된 NIS 도메인을 오버라이드할 수 있게 합니다. 지정하지 않으면, 조회는 기본 NIS 도메인에서 이루어집니다.

nis.**cat**(*mapname*, *domain=default_domain*)
`match(key, mapname)==value`가 되도록 *key*를 *value*에 매핑하는 딕셔너리를 반환합니다. 딕셔너리의 키와 값은 모두 임의의 바이트 배열입니다.

*mapname*이 다른 이름의 별칭인지 먼저 검사합니다.

domain 인자는 조회에 사용된 NIS 도메인을 오버라이드할 수 있게 합니다. 지정하지 않으면, 조회는 기본 NIS 도메인에서 이루어집니다.

nis.**maps**(*domain=default_domain*)
유효한 모든 맵 리스트를 반환합니다.

domain 인자는 조회에 사용된 NIS 도메인을 오버라이드할 수 있게 합니다. 지정하지 않으면, 조회는 기본 NIS 도메인에서 이루어집니다.

nis.**get_default_domain**()
시스템 기본 NIS 도메인을 반환합니다.

nis 모듈은 다음 예외를 정의합니다:

exception nis.**error**
NIS 함수가 에러 코드를 반환할 때 발생하는 에러.

36.13 syslog — 유닉스 syslog 라이브러리 루틴

이 모듈은 유닉스 syslog 라이브러리 루틴에 대한 인터페이스를 제공합니다. syslog 기능에 대한 자세한 설명은 유닉스 매뉴얼 페이지를 참조하십시오.

이 모듈은 시스템 syslog 계열의 루틴을 감쌉니다. syslog 서버와 통신할 수 있는 순수한 파이썬 라이브러리는 *logging.handlers* 모듈에서 SysLogHandler로 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.

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.

버전 3.2에서 변경: 이전 버전에서는, 키워드 인자가 허용되지 않았고, *ident*가 필수였습니다. *ident*의 기본값은 시스템 라이브러리에 따라 달랐으며, 종종 파이썬 프로그램 파일 이름 대신 python이었습니다.

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.

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.

모듈은 다음 상수를 정의합니다:

우선순위 수준 (높음에서 낮음 순서): LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG.

시설: LOG_KERN, LOG_USER, LOG_MAIL, LOG_DAEMON, LOG_AUTH, LOG_LPR, LOG_NEWS, LOG_UUCP, LOG_CRON, LOG_SYSLOG, LOG_LOCAL0 ~ LOG_LOCAL7 및 <syslog.h>에 정의되었으면, LOG_AUTHPRIV.

로그 옵션: LOG_PID, LOG_CONS, LOG_NDELAY 및 <syslog.h>에 정의되었으면, LOG_ODELAY, LOG_NOWAIT 및 LOG_PERROR.

36.13.1 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 37

대체된 모듈

이 장에서 설명하는 모듈은 폐지되었으며 하위 호환성을 위해서만 유지됩니다. 다른 모듈에 의해 대체되었습니다.

37.1 optparse — Parser for command line options

Source code: Lib/optparse.py

버전 3.2부터 폐지: The *optparse* module is deprecated and will not be developed further; development will continue with the *argparse* module.

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old *getopt* module. *optparse* uses a more declarative style of command-line parsing: you create an instance of *OptionParser*, populate it with options, and parse the command line. *optparse* allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using *optparse* in a simple script:

```
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()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options:

```
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
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

37.1.1 Background

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

Terminology

argument a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

option an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)

- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting VMS, MS-DOS, and/or Windows.

option argument an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, *optparse* does not support this feature.

positional argument something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

37.1.2 Tutorial

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable—for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *optparse* defaults to `store`.

The store action

The most common option action is `store`, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to `parse_args()`, `options.filename` is `"foo.txt"`.

Some other option types supported by *optparse* are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print 42.

If you don't specify a type, optparse assumes string. Combined with the fact that the default action is store, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, optparse figures out a sensible default from the option strings: if the first long option string is --foo-bar, then the default destination is foo_bar. If there are no long option strings, optparse looks at the first short option string: the default destination for -f is f.

optparse also includes the built-in complex type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. optparse supports them with two separate actions, store_true and store_false. For example, you might have a verbose flag that is turned on with -v and off with -q:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When optparse encounters -v on the command line, it sets options.verbose to True; when it encounters -q, options.verbose is set to False.

Other actions

Some other actions supported by optparse are:

"store_const" store a constant value

"append" append this option's argument to a list

"count" increment a counter by one

"callback" call a specified function

These are covered in section *Reference Guide*, Reference Guide and section *Option Callbacks*.

Default values

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to None. This is usually fine, but sometimes you want more control. optparse lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want optparse to set verbose to True unless -q is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling `parse_args()`:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
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]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
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]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message:

- the script defines its own usage message:

```
usage = "usage: %prog [options] arg1 arg2"
```

  optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

  If you don't supply a usage string, *optparse* uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping—*optparse* takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-generated help message, e.g. for the "mode" option:

```
-m MODE, --mode=MODE
```

  Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically-generated option description:

```
-f FILE, --filename=FILE
```

  This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include `%default` in the help string—*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup*:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

where

- parser is the *OptionParser* instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from `OptionContainer` (like *OptionParser*) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy:

```
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)
```

This would result in the following help output:

```
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.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
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)
```

that results in the following output:

```
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.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  -g                    Group option.

Debug Options:
  -d, --debug           Print debug information
  -s, --sql             Print all SQL statements executed
  -e                    Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

`OptionParser.`**`get_option_group`**(*opt_str*)
: Return the *OptionGroup* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *OptionGroup*, return `None`.

Printing a version string

Similar to the brief usage string, *optparse* can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, *optparse* automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

`OptionParser.`**`print_version`**(*file=None*)
: Print the version message for the current program (`self.version`) to *file* (default stdout). As with *print_usage()*, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**()
: Same as *print_version()* but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that *optparse* has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like:

```
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()
```

37.1.3 Reference Guide

Creating the parser

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (**default: "%prog [options]"**) The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (**default: []**) A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use *add_option()* after creating the parser instead.

option_class (**default: optparse.Option**) Class to use when adding options to the parser in *add_option()*.

version (**default: None**) A version string to print when the user supplies a version option. If you supply a true value for `version`, *optparse* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (**default: "error"**) Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

description (**default: None**) A paragraph of text giving a brief overview of your program. *optparse* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (**default: a new IndentedHelpFormatter**) An instance of optparse.HelpFormatter that will be used for printing help text. *optparse* provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (**default: True**) If true, *optparse* will add a help option (with option strings `-h` and `--help`) to the parser.

prog The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (**default: None**) A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using *OptionParser.add_option()*, as shown in section *Tutorial*. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
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()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of *optparse* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an `Option` instance is with the `add_option()` method of *OptionParser*.

`OptionParser.`**`add_option`**`(`*option*`)`
`OptionParser.`**`add_option`**`(`**opt_str*, *attr*=*value*, ...`)`

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *action*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *optparse* raises an `OptionError` exception explaining your mistake.

An option's *action* determines what *optparse* does when it encounters this option on the command-line. The standard option actions hard-coded into *optparse* are:

`"store"` store this option's argument (default)

`"store_const"` store a constant value

`"store_true"` store `True`

`"store_false"` store `False`

`"append"` append this option's argument to a list

`"append_const"` append a constant value to a list

`"count"` increment a counter by one

`"callback"` call a specified function

`"help"` print a usage message including all options and the documentation for them

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *type* and *dest* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *optparse* always creates a special object for this, conventionally called `options` (it happens to be an instance of `optparse.Values`). Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *optparse* does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises `OptionError`.

`Option.`**`action`**
(default: `"store"`)

Determines *optparse*'s behaviour when this option is seen on the command line; the available options are documented *here*.

`Option.`**`type`**
(default: `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`); the available option types are documented *here*.

`Option.`**`dest`**
(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it: *dest* names an attribute of the `options` object that *optparse* builds as it parses the command line.

`Option.`**`default`**
The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

`Option.`**`nargs`**
(default: 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

Option.**const**
: For actions that store a constant value, the constant value to store.

Option.**choices**
: For options of type `"choice"`, the list of strings the user may choose from.

Option.**callback**
: For options with action `"callback"`, the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

Option.**callback_args**
Option.**callback_kwargs**
: Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.

Option.**help**
: Help text to print for this option when listing all available options after the user supplies a *help* option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option.**metavar**
: (default: derived from option strings)

  Stand-in for the option argument(s) to use when printing help text. See section *Tutorial* for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour; a few have required attributes, which you must specify for any option using that action.

- `"store"` [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

  If *choices* is supplied (a list or tuple of strings), the type defaults to `"choice"`.

  If *type* is not supplied, it defaults to `"string"`.

  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., `-f` implies `f`).

  Example:

  ```
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```

  As it parses the command line

  ```
  -f foo.txt -p 1 -3.5 4 -fbar.txt
  ```

  optparse will set

  ```
  options.f = "foo.txt"
  options.point = (1.0, -3.5, 4.0)
  options.f = "bar.txt"
  ```

- "store_const" [required: *const*; relevant: *dest*]

  The value *const* is stored in *dest*.

  Example:

```
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")
```

  If --noisy is seen, *optparse* will set

```
options.verbose = 2
```

- "store_true" [relevant: *dest*]

  A special case of "store_const" that stores True to *dest*.

- "store_false" [relevant: *dest*]

  Like "store_true", but stores False.

  Example:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

  The defaults for *type* and *dest* are the same as for the "store" action.

  Example:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  If -t3 is seen on the command-line, *optparse* does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

  If, a little later on, --tracks=4 is seen, it does:

```
options.tracks.append(int("4"))
```

  The append action calls the append method on the current value of the option. This means that any default value specified must have an append method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> 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" [required: *const*; relevant: *dest*]

  Like "store_const", but the value *const* is appended to *dest*; as with "append", *dest* defaults to None, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: *dest*]

  Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

  Example:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  The first time -v is seen on the command line, *optparse* does the equivalent of:

```
options.verbosity = 0
options.verbosity += 1
```

  Every subsequent occurrence of -v results in

```
options.verbosity += 1
```

- "callback" [required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  Call the function specified by *callback*, which is called as

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  See section *Option Callbacks* for more detail.

- "help"

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the usage string passed to OptionParser's constructor and the *help* string passed to every option.

  If no *help* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value optparse.SUPPRESS_HELP.

  optparse automatically adds a *help* option to all OptionParsers, so you do not normally need to create one.

  Example:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
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)
```

  If *optparse* sees either -h or --help on the command line, it will print something like the following help message to stdout (assuming sys.argv[0] is "foo.py"):

```
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
```

After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *help* options, you will rarely create `version` options, since *optparse* automatically adds them when needed.

Standard option types

optparse has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling *int()* with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.

Parsing arguments

The whole point of creating and populating an OptionParser is to call its `parse_args()` method:

```
(options, args) = parser.parse_args(args=None, values=None)
```

where the input parameters are

args the list of arguments to process (default: `sys.argv[1:]`)

values an `optparse.Values` object to store option arguments in (default: a new instance of `Values`) – if you give an existing object, the option defaults will not be initialized on it

and the return values are

options the same object that was passed in as `values`, or the optparse.Values instance created by *optparse*

args the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated *`setattr()`* calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

`OptionParser.`**`disable_interspersed_args`**`()`
Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *`optparse`* normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *`disable_interspersed_args()`*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.`**`enable_interspersed_args`**`()`
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

`OptionParser.`**`get_option`**`(`*opt_str*`)`
Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

`OptionParser.`**`has_option`**`(`*opt_str*`)`
Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

`OptionParser.`**`remove_option`**`(`*opt_str*`)`
If the *`OptionParser`* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *`OptionParser`*, raises *`ValueError`*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *`optparse`* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

"error" (default) assume option conflicts are a programming error and raise OptionConflictError

"resolve" resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously-added option is already using the -n option string. Since conflict_handler is "resolve", it resolves the situation by removing -n from the earlier option's list of option strings. Now --dry-run is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original -n/--dry-run option is no longer accessible, so *optparse* removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling destroy() on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

`OptionParser.`**`set_usage`**(*usage*)
Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**(*file=None*)
Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**()
Same as *print_usage()* but returns the usage string instead of printing it.

`OptionParser.`**`set_defaults`**(*dest=value, ...*)
Set default values for several option destinations at once. Using *set_defaults()* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use *set_defaults()*:

```
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")
```

37.1.4 Option Callbacks

When *optparse*'s built-in actions and types aren't quite enough for your needs, you have two choices: extend *optparse* or define a callback option. Extending *optparse* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is callback, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

callback is a function (or other callable object), so you must have already defined my_callback() when you create this callback option. In this simple case, *optparse* doesn't even know if -c takes any arguments, which usually means that the option takes no arguments—the mere presence of -c on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type has its usual meaning: as with the "store" or "append" actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs also has its usual meaning: if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args a tuple of extra positional arguments to pass to the callback

callback_kwargs a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option is the Option instance that's calling the callback

opt_str is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, opt_str will be the full, canonical option string—e.g. if the user puts --foo on the command-line as an abbreviation for --foobar, then opt_str will be "--foobar".)

value is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of value will be the type implied by the option's type. If *type* for this option is None (no argument expected), then value will be None. If *nargs* > 1, value will be a tuple of values of the appropriate type.

parser is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

parser.largs the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify parser.largs, e.g. by adding more arguments to it. (This list will become args, the second return value of parse_args().)

parser.rargs the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
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")
```

Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
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')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
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")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a `"store"` or `"append"` option: if you define *type*, then the option takes one argument that must be convertible to that type; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard `"store"` action:

```
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")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
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:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        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)
```

37.1.5 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s `Option` class. This class has a couple of attributes that define *optparse*'s types: *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**
A tuple of type names; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.`**`TYPE_CHECKER`**
A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.

Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions:

"store" actions actions that result in *optparse* storing a value to an attribute of the current OptionValues instance; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets: some default "store" actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

`Option.`**ACTIONS**
All actions must be listed in ACTIONS.

`Option.`**STORE_ACTIONS**
"store" actions are additionally listed here.

`Option.`**TYPED_ACTIONS**
"typed" actions are additionally listed here.

`Option.`**ALWAYS_TYPED_ACTIONS**
Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, `"string"`, to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in *ALWAYS_TYPED_ACTIONS* as well.
- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard *optparse* actions.
- `values` is an instance of the optparse_parser.Values class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially *getattr()* with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns 'value. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it's needed.

37.2 `imp` — Access the import internals

Source code: Lib/imp.py

버전 3.4부터 폐지: The *imp* module is deprecated in favor of *importlib*.

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions:

`imp.`**`get_magic`**`()`
Return the magic string value used to recognize byte-compiled code files (`.pyc` files). (This value may be different for each Python version.)

버전 3.4부터 폐지: Use *importlib.util.MAGIC_NUMBER* instead.

`imp.`**`get_suffixes`**`()`
Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form `(suffix, mode, type)`, where *suffix* is a string to be appended to the module name to form the filename to search for, *mode* is the mode string to pass to the built-in *open()* function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and *type* is the file type, which has one of the values *PY_SOURCE*, *PY_COMPILED*, or *C_EXTENSION*, described below.

버전 3.3부터 폐지: Use the constants defined on *importlib.machinery* instead.

imp.**find_module**(*name*[, *path*])

Try to find the module *name*. If *path* is omitted or `None`, the list of directory names given by `sys.path` is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (*C_BUILTIN*), then a frozen module (*PY_FROZEN*), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, *path* must be a list of directory names; each directory is searched for files with any of the suffixes returned by *get_suffixes()* above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple `(file, pathname, description)`:

file is an open *file object* positioned at the beginning, *pathname* is the pathname of the file found, and *description* is a 3-element tuple as contained in the list returned by *get_suffixes()* describing the kind of module found.

If the module is built-in or frozen then *file* and *pathname* are both `None` and the *description* tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, *ImportError* is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, *file* is `None`, *pathname* is the package path and the last item in the *description* tuple is *PKG_DIRECTORY*.

This function does not handle hierarchical module names (names containing dots). In order to find *P.M*, that is, submodule *M* of package *P*, use *find_module()* and *load_module()* to find and load package *P*, and then use *find_module()* with the *path* argument set to `P.__path__`. When *P* itself has a dotted name, apply this recipe recursively.

버전 3.3부터 폐지: Use *importlib.util.find_spec()* instead unless Python 3.3 compatibility is required, in which case use *importlib.find_loader()*. For example usage of the former case, see the *Examples* section of the *importlib* documentation.

imp.**load_module**(*name*, *file*, *pathname*, *description*)

Load a module that was previously found by *find_module()* (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it will reload the module! The *name* argument indicates the full module name (including the package name, if this is a submodule of a package). The *file* argument is an open file, and *pathname* is the corresponding file name; these can be `None` and `''`, respectively, when the module is a package or not being loaded from a file. The *description* argument is a tuple, as would be returned by *get_suffixes()*, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually *ImportError*) is raised.

Important: the caller is responsible for closing the *file* argument, if it was not `None`, even when an exception is raised. This is best done using a `try` ⋯ `finally` statement.

버전 3.3부터 폐지: If previously used in conjunction with *imp.find_module()* then consider using *importlib.import_module()*, otherwise use the loader returned by the replacement you chose for *imp.find_module()*. If you called *imp.load_module()* and related functions directly with file path arguments then use a combination of *importlib.util.spec_from_file_location()* and *importlib.util.module_from_spec()*. See the *Examples* section of the *importlib* documentation for details of the various approaches.

imp.**new_module**(*name*)

Return a new empty module object called *name*. This object is *not* inserted in `sys.modules`.

버전 3.4부터 폐지: Use *importlib.util.module_from_spec()* instead.

imp.**reload**(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try

out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When `reload(module)` is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for *sys*, *__main__* and *builtins*. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` … `import` …, calling *reload()* for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module*.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

버전 3.3에서 변경: Relies on both `__name__` and `__loader__` being defined on the module being reloaded instead of just `__name__`.

버전 3.4부터 폐지: Use *importlib.reload()* instead.

The following functions are conveniences for handling **PEP 3147** byte-compiled file paths.

버전 3.2에 추가.

`imp.`**`cache_from_source`**(*path*, *debug_override=None*)

Return the **PEP 3147** path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see *get_tag()*; if `sys.implementation.cache_tag` is not defined then *NotImplementedError* will be raised). By passing in `True` or `False` for *debug_override* you can override the system's value for `__debug__`, leading to optimized bytecode.

path need not exist.

버전 3.3에서 변경: If `sys.implementation.cache_tag` is `None`, then *NotImplementedError* is raised.

버전 3.4부터 폐지: Use *importlib.util.cache_from_source()* instead.

버전 3.5에서 변경: The *debug_override* parameter no longer creates a `.pyo` file.

imp.**source_from_cache**(*path*)
Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

버전 3.3에서 변경: Raise *NotImplementedError* when `sys.implementation.cache_tag` is not defined.

버전 3.4부터 폐지: Use *importlib.util.source_from_cache()* instead.

imp.**get_tag**()
Return the **PEP 3147** magic tag string matching this version of Python's magic number, as returned by *get_magic()*.

버전 3.4부터 폐지: Use `sys.implementation.cache_tag` directly starting in Python 3.3.

The following functions help interact with the import system's internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

imp.**lock_held**()
Return `True` if the global import lock is currently held, else `False`. On platforms without threads, always return `False`.

On platforms with threads, a thread executing an import first holds a global import lock, then sets up a per-module lock for the rest of the import. This blocks other threads from importing the same module until the original import completes, preventing other threads from seeing incomplete module objects constructed by the original thread. An exception is made for circular imports, which by construction have to expose an incomplete module object at some point.

버전 3.3에서 변경: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

버전 3.4부터 폐지.

imp.**acquire_lock**()
Acquire the interpreter's global import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

버전 3.3에서 변경: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

버전 3.4부터 폐지.

imp.**release_lock**()
Release the interpreter's global import lock. On platforms without threads, this function does nothing.

버전 3.3에서 변경: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

버전 3.4부터 폐지.

The following constants with integer values, defined in this module, are used to indicate the search result of *find_module()*.

imp.**PY_SOURCE**
The module was found as a source file.

버전 3.3부터 폐지.

imp.**PY_COMPILED**
The module was found as a compiled code object file.

버전 3.3부터 폐지.

imp.**C_EXTENSION**
The module was found as dynamically loadable shared library.

버전 3.3부터 폐지.

imp.**PKG_DIRECTORY**
The module was found as a package directory.

버전 3.3부터 폐지.

imp.**C_BUILTIN**
The module was found as a built-in module.

버전 3.3부터 폐지.

imp.**PY_FROZEN**
The module was found as a frozen module.

버전 3.3부터 폐지.

class imp.**NullImporter**(*path_string*)
The *NullImporter* type is a **PEP 302** import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises *ImportError*. Otherwise, a *NullImporter* instance is returned.

Instances have only one method:

find_module(*fullname*[, *path*])
This method always returns None, indicating that the requested module could not be found.

버전 3.3에서 변경: None is inserted into sys.path_importer_cache instead of an instance of *NullImporter*.

버전 3.4부터 폐지: Insert None into sys.path_importer_cache instead.

37.2.1 Examples

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since *find_module()* has been extended and *load_module()* has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

CHAPTER 38

문서로 만들어지지 않은 모듈

여기, 현재 문서로 만들어지지 않았지만, 문서로 만들어야 하는 모듈의 목록이 있습니다. 이것들의 문서를 기고해 주십시오! (이메일을 통해 docs@python.org 로 보내주십시오.)

이 장의 아이디어와 원래 내용은 Fredrik Lundh의 글에서 가져온 것입니다; 이 장의 구체적인 내용은 상당히 개정되었습니다.

38.1 플랫폼 특정 모듈

이 모듈은 *os.path* 모듈을 구현하는 데 사용되며, 이 언급 이상의 것들이 설명되지 않았습니다. 문서로 만들 필요가 거의 없습니다.

ntpath — Win32 와 Win64 플랫폼에서 *os.path*를 구현합니다.

posixpath — POSIX에서 *os.path*를 구현합니다.

APPENDIX A

용어집

>>> 대화형 셸의 기본 파이썬 프롬프트. 인터프리터에서 대화형으로 실행될 수 있는 코드 예에서 자주 볼 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.

2to3 파이썬 2.x 코드를 파이썬 3.x 코드로 변환하려고 시도하는 도구인데, 소스를 구문 분석하고 구문 분석 트리를 탐색해서 감지할 수 있는 대부분의 비호환성을 다룹니다.

2to3 는 표준 라이브러리에서 *lib2to3* 로 제공됩니다; 독립적으로 실행할 수 있는 스크립트는 `Tools/scripts/2to3` 로 제공됩니다. *2to3 - 파이썬 2에서 파이썬 3으로 자동 코드 변환*을 보세요.

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 타이핑을 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.

annotation (어노테이션) 관습에 따라 형 힌트 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 반환 값과 연결된 레이블입니다.

지역 변수의 어노테이션은 실행 시간에 액세스할 수 없지만, 전역 변수, 클래스 속성 및 함수의 어노테이션은 각각 모듈, 클래스, 함수의 `__annotations__` 특수 어트리뷰트에 저장됩니다.

이 기능을 설명하는 변수 어노테이션, 함수 어노테이션, **PEP 484**, **PEP 526**을 참조하세요.

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) 비동기 제너레이터 함수가 만드는 객체.

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, 그 위치의 (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.

attribute (어트리뷰트) 점표현식을 사용하는 이름으로 참조되는 객체와 결합한 값. 예를 들어, 객체 *o*가 어트리뷰트 *a*를 가지면, *o.a*처럼 참조됩니다.

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.

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.

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.

coercion (코어션) 같은 형의 두 인자를 수반하는 연산이 일어나는 동안, 한 형의 인스턴스를 다른 형으로 묵시적으로 변환하는 것. 예를 들어, `int(3.15)` 는 실수를 정수 `3`으로 변환합니다. 하지만, `3+4.5` 에서, 각 인자는 다른 형이고 (하나는 int, 다른 하나는 float), 둘을 더하기 전에 같은 형으로 변환해야 합니다. 그렇지 않으면 *TypeError*를 일으킵니다. 코어션 없이는, 호환되는 형들조차도 프로그래머가 같은 형으로 정규화해주어야 합니다, 예를 들어, 그냥 `3+4.5` 하는 대신 `float(3)+4.5`.

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 manager (컨텍스트 관리자) `__enter__()` 와 `__exit__()` 메서드를 정의함으로써 `with` 문에서 보이는 환경을 제어하는 객체. **PEP 343**으로 도입되었습니다.

context variable (컨텍스트 변수) 컨텍스트에 따라 다른 값을 가질 수 있는 변수. 이는 각 실행 스레드가 변수에 대해 다른 값을 가질 수 있는 스레드-로컬 저장소와 비슷합니다. 그러나, 컨텍스트 변수를 통해, 하나의 실행 스레드에 여러 컨텍스트가 있을 수 있으며 컨텍스트 변수의 주 용도는 동시성 비동기 태스크에서 변수를 추적하는 것입니다. *contextvars*를 참조하십시오.

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) Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492**.

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.

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(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의와 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, 슈퍼클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 descriptors에 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 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 변화를 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. 인트로스팩션을 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.

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.

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'` 나 `'F'` 를 앞에 붙인 문자열 리터럴들을 흔히 "f-문자열"이라고 부르는데, 포맷 문자열 리터럴 의 줄임말입니다. **PEP 498** 을 보세요.

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.

finder (파인더) 임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

파이썬 3.3. 이후로, 두 종류의 파인더가 있습니다: *sys.meta_path* 와 함께 사용하는 메타 경로 파인더와 *sys.path_hooks* 과 함께 사용하는 경로 엔트리 파인더.

더 자세한 내용은 **PEP 302**, **PEP 420**, **PEP 451** 에 나옵니다.

floor division (정수 나눗셈) 가장 가까운 정수로 내림하는 수학적 나눗셈. 정수 나눗셈 연산자는 `//` 다. 예를 들어, 표현식 `11 // 4` 의 값은 2가 되지만, 실수 나눗셈은 `2.75`를 돌려줍니다. `(-11) // 4` 가 `-2.75` 를 내림 한 `-3`이 됨에 유의해야 합니다. **PEP 238**을 보세요.

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.

__future__ 프로그래머가 현재 인터프리터와 호환되지 않는 새 언어 기능들을 활성화할 수 있도록 하는 가상 모듈.

__future__ 모듈을 임포트하고 그 변수들의 값들을 구해서, 새 기능이 언제 처음으로 언어에 추가되었고, 언제부터 그것이 기본이 되는지 볼 수 있습니다:

```
>>> 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, 그 위치의 (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.

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.

(훨씬 더 미세하게 공유 데이터를 잠그는) "스레드에 자유로운(free-threaded)" 인터프리터를 만들고자 하는 과거의 노력은 성공적이지 못했는데, 흔한 단일 프로세서 경우의 성능 저하가 심하기 때문입니다. 이 성능 이슈를 극복하는 것은 구현을 훨씬 복잡하게 만들어서 유지 비용이 더 들어갈 것으로 여겨지고 있습니다.

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 환경 (Integrated Development Environment). IDLE은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

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 들여다보는 매우 강력한 방법입니다 (`help(x)`를 기억하세요).

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.

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* 모듈은 세 개의 키 함수 생성자를 제공합니다: *attrgetter()*, *itemgetter()*, *methodcaller()*. 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.

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) 모듈을 로드하는 객체. `load_module()` 이라는 이름의 메서드를 정의해야 합니다. 로더는 보통 파인더 가 돌려줍니다. 자세한 내용은 **PEP 302** 를, 추상 베이스 클래스 는 *importlib.abc.Loader* 를 보세요.

magic method (매직 메서드) 특수 메서드 의 비공식적인 비슷한 말.

mapping (매핑) 임의의 키 조회를 지원하고 *Mapping* 이나 *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 The Python 2.3 Method Resolution Order를 보면 됩니다.

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.

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) The term "named tuple" applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by *time.localtime()* and *os.stat()*. Another example is *sys.float_info*:

```
>>> 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
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from *tuple* and that defines named fields. Such a class can be written by hand or it can be created with the factory function *collections.namedtuple()*. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.

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 **PEP 420** 패키지. 이름 공간 패키지는 물리적인 실체가 없을 수도 있고, 특히 `__init__.py` 파일이 없으므로 정규 패키지 와는 다릅니다.

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 클래스 의 최종적인 베이스 클래스입니다.

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. 파이썬은 위치-전용 매개변수를 정의하는 문법을 갖고 있지 않습니다. 하지만, 어떤 매장 함수들은 위치-전용 매개변수를 갖습니다 (예를 들어, *abs()*).

- 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_hook 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* 객체이거나 *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. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소입니다. *sys* 모듈은 특정 객체의 참조 횟수를 돌려주는 *getrefcount()* 을 정의합니다.

regular package (정규 패키지) `__init__.py` 파일을 포함하는 디렉터리와 같은 전통적인 패키지.

이름 공간 패키지 도 보세요.

__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 불변 키를 사용하기 때문에 시퀀스가 아니라 매핑으로 취급된다는 것에 주의해야 합니다.

collections.abc.Sequence 추상 베이스 클래스는 `__getitem__()` 과 `__len__()` 을 넘어서 훨씬 풍부한 인터페이스를 정의하는데, `count()`, `index()`, `__contains__()`, `__reversed__()` 를 추가합니다. 이 확장된 인터페이스를 구현한 형을 `register()` 를 사용해서 명시적으로 등록할 수 있습니다.

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 (서브스크립트) 표기법은 내부적으로 *slice* 객체를 사용합니다.

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.

statement (문장) 문장은 스위트 (코드의 "블록(block)") 를 구성하는 부분입니다. 문장은 표현식 이거나 키워드를 사용하는 여러 가지 구조물 중의 하나입니다. 가령 `if`, `while`, `for`.

text encoding (텍스트 인코딩) 유니코드 문자열을 바이트열로 인코딩하는 코덱.

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.

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:

```
from typing import List, Tuple

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:

```
from typing import List, Tuple

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 사항이며 파이썬에서 강제되지는 않습니다. 하지만, 정적 형 분석 도구에 유용하며 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.

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.

Zen of Python (파이썬 젠) 파이썬 디자인 원리와 철학들의 목록인데, 인어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 "`import this`"를 입력하면 보입니다.

APPENDIX B

이 설명서에 관하여

이 설명서는 reStructuredText 소스에서 만들어진 것으로, 파이썬 설명서를 위해 특별히 제작된 문서 처리기인 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 작가;
- reStructuredText와 Docutils 스위트를 만드는 Docutils 프로젝트.
- Fredrik Lundh, 그의 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

파이썬은 ABC라는 언어의 후계자로서 네덜란드의 Stichting Mathematisch Centrum (CWI, https://www.cwi.nl/ 참조)의 Guido van Rossum에 의해 1990년대 초반에 만들어졌습니다. 파이썬에는 다른 사람들의 많은 공헌이 포함되었지만, Guido는 파이썬의 주요 저자로 남아 있습니다.

1995년, Guido는 Virginia의 Reston에 있는 Corporation for National Research Initiatives(CNRI, https://www.cnri.reston.va.us/ 참조)에서 파이썬 작업을 계속했고, 이곳에서 여러 버전의 소프트웨어를 출시했습니다.

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 (now Zope Corporation; see https://www.zope.org/). 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 is a sponsoring member of the PSF.

모든 파이썬 배포판은 공개 소스입니다 (공개 소스 정의에 대해서는 https://opensource.org/를 참조하십시오). 역사적으로, 대부분 (하지만 전부는 아닙니다) 파이썬 배포판은 GPL과 호환됩니다; 아래의 표는 다양한 배포판을 요약한 것입니다.

배포판	파생된 곳	해	소유자	GPL 호환?
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	no
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: GPL과 호환된다는 것은 우리가 GPL로 파이썬을 배포한다는 것을 의미하지는 않습니다. 모든 파이썬 라이센스는 GPL과 달리 여러분의 변경을 공개 소스로 만들지 않고 수정된 버전을 배포할 수 있게 합니다. GPL 호환 라이센스는 파이썬과 GPL 하에 발표된 다른 소프트웨어를 결합할 수 있게 합니다; 다른 것들은 그렇지 않습니다.

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.17

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.7.17 software 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 3.7.17 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.7.17 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 3.7.17 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
   3.7.17.

4. PSF is making Python 3.7.17 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 3.7.17 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.17
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.17, 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
   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 3.7.17, 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   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
   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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   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.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

_random 모듈은 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 에서 내려받은 코드에 기반한 코드를 포함합니다. 다음은 원래 코드의 주석을 그대로 옮긴 것입니다:

```
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).
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
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. 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

socket 모듈은 getaddrinfo() 와 getnameinfo() 함수를 사용합니다. 이들은 WIDE Project, http://www.wide.ad.jp/, 에서 온 별도 소스 파일로 코딩되어 있습니다.

```
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.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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
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

*asynchat*과 *asyncore* 모듈은 다음과 같은 주의 사항을 포함합니다:

```
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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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
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

uu 모듈은 다음과 같은 주의 사항을 포함합니다:

```
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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.

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

test_epoll 모듈은 다음과 같은 주의 사항을 포함합니다:

```
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
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/

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

C double과 문자열 간의 변환을 위한 C 함수 dtoa 와 strtod 를 제공하는 파일 Python/dtoa.c 는 현재 http://www.netlib.org/fp/ 에서 얻을 수 있는 David M. Gay의 같은 이름의 파일에서 파생되었습니다. 2009년 3월 16일에 받은 원본 파일에는 다음과 같은 저작권 및 라이센스 공지가 포함되어 있습니다:

```
/****************************************************************
 *
 * 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

모듈 *hashlib*, *posix*, *ssl*, *crypt* 는 운영 체제가 사용할 수 있게 하면 추가의 성능을 위해 OpenSSL 라이브러리를 사용합니다. 또한, 윈도우와 맥 OS X 파이썬 설치 프로그램은 OpenSSL 라이브러리 사본을 포함할 수 있으므로, 여기에 OpenSSL 라이센스 사본을 포함합니다:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL 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. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED 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 OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    * 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.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * 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 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. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``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.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

pyexpat 확장은 빌드를 --with-system-expat 로 구성하지 않는 한, 포함된 expat 소스 사본을 사용하여 빌드됩니다:

```
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
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

_ctypes 확장은 빌드를 --with-system-libffi 로 구성하지 않는 한, 포함된 libffi 소스 사본을 사용하여 빌드됩니다:

```
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.

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

_decimal 모듈은 빌드를 --with-system-libmpdec 로 구성하지 않는 한, 포함된 libmpdec 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 2008-2016 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.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.
```

APPENDIX D

저작권

파이썬과 이 설명서는:

전체 라이센스 및 사용 권한 정보는 역사와 라이센스 에서 제공합니다.

Bibliography

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009. The third edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.

[C99] ISO/IEC 9899:1999. "Programming languages – C." A public draft of this standard is available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

Python 모듈 목록

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

W

X

Z

Non-alphabetical

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z