The Python/C API

릴리스 *3.14.0rc3*

Guido van Rossum and the Python development team

10월 01, 2025

Contents

1 소개 **3**
- 1.1 Language version compatibility . . . 3
- 1.2 코딩 표준 . . . 3
- 1.3 인클루드 파일 . . . 3
- 1.4 유용한 매크로들 . . . 4
- 1.5 객체, 형 그리고 참조 횟수 . . . 6
- 1.6 예외 . . . 10
- 1.7 파이썬 임베딩하기 . . . 12
- 1.8 디버깅 빌드 . . . 12
- 1.9 Recommended third party tools . . . 13

2 C API Stability **15**
- 2.1 Unstable C API . . . 15
- 2.2 안정적인 응용 프로그램 바이너리 인터페이스 . . . 15
- 2.3 Platform Considerations . . . 17
- 2.4 Contents of Limited API . . . 17

3 매우 고수준 계층 **43**

4 참조 횟수 **49**

5 예외 처리 **53**
- 5.1 인쇄와 지우기 . . . 53
- 5.2 예외 발생시키기 . . . 54
- 5.3 경고 발행하기 . . . 57
- 5.4 에러 표시기 조회하기 . . . 57
- 5.5 시그널 처리하기 . . . 61
- 5.6 예외 클래스 . . . 62
- 5.7 예외 객체 . . . 62
- 5.8 유니코드 예외 객체 . . . 63
- 5.9 재귀 제어 . . . 64
- 5.10 Exception and warning types . . . 65

6 Defining extension modules **73**
- 6.1 Multiple module instances . . . 73
- 6.2 Initialization function . . . 74
- 6.3 Multi-phase initialization . . . 75
- 6.4 Legacy single-phase initialization . . . 75

7 유틸리티 **77**
- 7.1 운영 체제 유틸리티 . . . 77

7.2 시스템 함수 . . . 80
7.3 프로세스 제어 . . . 82
7.4 모듈 임포트 하기 . . . 82
7.5 데이터 마샬링 지원 . . . 86
7.6 인자 구문 분석과 값 구축 . . . 87
7.7 문자열 변환과 포매팅 . . . 96
7.8 PyHash API . . . 98
7.9 리플렉션 . . . 99
7.10 코덱 등록소와 지원 함수 . . . 100
7.11 PyTime C API . . . 102
7.12 Support for Perf Maps . . . 103

8 추상 객체 계층 105
8.1 객체 프로토콜 . . . 105
8.2 호출 프로토콜 . . . 115
8.3 숫자 프로토콜 . . . 120
8.4 시퀀스 프로토콜 . . . 123
8.5 매핑 프로토콜 . . . 125
8.6 이터레이터 프로토콜 . . . 126
8.7 버퍼 프로토콜 . . . 127

9 구상 객체 계층 135
9.1 기본 객체 . . . 135
9.2 숫자 객체 . . . 142
9.3 시퀀스 객체 . . . 156
9.4 컨테이너 객체 . . . 185
9.5 함수 객체 . . . 192
9.6 기타 객체 . . . 202

10 초기화, 파이널리제이션 및 스레드 229
10.1 파이썬 초기화 전 . . . 229
10.2 전역 구성 변수 . . . 230
10.3 인터프리터 초기화와 파이널리제이션 . . . 233
10.4 프로세스 전체 매개 변수 . . . 236
10.5 스레드 상태와 전역 인터프리터 록 . . . 239
10.6 서브 인터프리터 지원 . . . 247
10.7 비동기 알림 . . . 250
10.8 프로파일링과 추적 . . . 251
10.9 Reference tracing . . . 253
10.10 고급 디버거 지원 . . . 253
10.11 스레드 로컬 저장소 지원 . . . 254
10.12 Synchronization Primitives . . . 256

11 파이썬 초기화 구성 259
11.1 PyInitConfig C API . . . 259
11.2 Configuration Options . . . 262
11.3 Runtime Python configuration API . . . 263
11.4 PyConfig C API . . . 265
11.5 Py_GetArgcArgv() . . . 285
11.6 Delaying main module execution . . . 285

12 메모리 관리 287
12.1 개요 . . . 287
12.2 Allocator Domains . . . 288
12.3 원시 메모리 인터페이스 . . . 288
12.4 메모리 인터페이스 . . . 289
12.5 객체 할당자 . . . 290
12.6 기본 메모리 할당자 . . . 292
12.7 메모리 할당자 사용자 정의 . . . 292

12.8 파이썬 메모리 할당자의 디버그 훅 . . . 294
12.9 pymalloc 할당자 . . . 295
12.10 mimalloc 할당자 . . . 296
12.11 tracemalloc C API . . . 296
12.12 예 . . . 296

13 객체 구현 지원 **299**
13.1 힙에 객체 할당하기 . . . 299
13.2 Object Life Cycle . . . 301
13.3 공통 객체 구조체 . . . 305
13.4 Type Object Structures . . . 314
13.5 순환 가비지 수집 지원 . . . 354

14 API와 ABI 버전 붙이기 **359**
14.1 Build-time version constants . . . 359
14.2 Run-time version . . . 359
14.3 Bit-packing macros . . . 360

15 Monitoring C API **361**

16 Generating Execution Events **363**
16.1 Managing the Monitoring State . . . 364

A 용어집 **369**

B 이 설명서에 관하여 **387**
B.1 파이썬 설명서의 공헌자들 . . . 387

C 역사와 라이센스 **389**
C.1 소프트웨어의 역사 . . . 389
C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관 . . . 390
C.3 포함된 소프트웨어에 대한 라이센스 및 승인 . . . 393

D 저작권 **409**

Bibliography **411**

색인 **413**

이 설명서는 확장 모듈을 작성하거나 파이썬을 내장하고자 하는 C와 C++ 프로그래머가 사용하는 API에 관해 설명합니다. 이 설명서와 쌍을 이루는 extending-index 는 확장 제작의 일반 원칙을 설명하지만, API 함수를 자세하게 설명하지는 않습니다.

CHAPTER 1

소개

파이썬의 애플리케이션 프로그래머용 인터페이스는 다양한 수준에서 C/C++ 프로그래머에게 파이썬 인터프리터에 대한 접근 방법을 제공합니다. 이 API는 C++에서도 동일하게 사용 가능하지만 간결함을 위해 보통 파이썬/C API 로 불립니다. 파이썬/C API를 사용하는 데에는 근본적으로 다른 두 가지 이유가 있습니다. 첫번째 이유는 특정한 목적을 위해 확장 모듈을 작성하기 위해서입니다; 이 확장 모듈들은 파이썬 인터프리터를 확장하는 C 모듈들입니다. 이것이 아마도 가장 흔한 용도일 것입니다. 두번째 이유는 파이썬을 더 큰 애플리케이션의 컴포넌트로 사용하기 위함입니다. 이 기술은 일반적으로 파이썬을 애플리케이션에 임베딩(*embedding*) 하는 것을 말합니다.

확장 모듈을 작성하는 것은 비교적 잘 다듬어진 과정으로, "쿡북" 접근법이 잘 통하며 프로세스를 다소 자동화하는 툴들도 존재합니다. 사람들은 파이썬이 존재한 초기부터 다른 애플리케이션에 파이썬을 임베드해왔으나 파이썬을 임베딩 하는 과정은 확장 모듈을 작성하는 것보다 복잡합니다.

많은 API 함수들은 파이썬을 임베딩하거나 확장하는 것에 무관하게 유용합니다. 더욱이 파이썬을 임베드하는 대부분의 애플리케이션은 커스텀 확장을 제공할 필요성이 있기 때문에 파이썬을 임베드하려고 시도하기 전에 확장을 작성하는 것에 친숙해지는 것이 좋습니다.

1.1 Language version compatibility

Python' s C API is compatible with C11 and C++11 versions of C and C++.

This is a lower limit: the C API does not require features from later C/C++ versions. You do *not* need to enable your compiler' s "c11 mode".

1.2 코딩 표준

CPython 에 포함하기 위해 C 코드를 작성하는 경우에는 **PEP 7** 에 정의된 지침과 표준을 따라야 합니다. 이 지침은 기여하고 있는 파이썬 버전과 상관없이 적용됩니다. 최종적으로 파이썬에 기여하는 것을 기대하지 않는 이상 이 규칙을 따르는 것은 제삼자 확장 모듈에는 필수가 아닙니다.

1.3 인클루드 파일

파이썬/C API를 사용하기 위한 모든 함수, 타입 그리고 매크로 정의는 다음 행에 의해 인클루드됩니다.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

이는 다음과 같은 표준 헤더를 인클루드하는것을 의미합니다: <stdio.h>, <string.h>, <errno.h>, <limits.h>, <assert.h> 그리고 <stdlib.h> (사용 가능한 경우).

> **참고**
>
> 파이썬은 일부 시스템의 표준 헤더에 영향을 미치는 전처리기 정의를 정의할 수 있으므로 표준 헤더를 인클루드하기 전에 Python.h 를 인클루드해야 합니다.
>
> Python.h 를 인클루드하기 전에 항상 PY_SSIZE_T_CLEAN 를 정의하는 것을 권장합니다. 이 매크로에 대한 자세한 사항은 인자 구문 분석과 값 구축 을 참조하십시오.

Python.h로 정의된 사용자에게 공개되는 모든 이름들은 (포함된 표준 헤더로 정의된 것은 제외) Py 또는 _Py 로 시작하는 이름을 가지고 있습니다. _Py 로 시작하는 이름들은 파이썬 구현에 의해 내부적으로 사용되며 확장 개발자들에 의해 사용되어서는 안됩니다. 구조체 멤버들은 이름에 접두사가 붙지 않습니다.

> **참고**
>
> 사용자 코드는 Py 또는 _Py 로 시작하는 이름들을 정의해서는 안됩니다. 이것은 읽는 사람을 혼란스럽게 하며 이러한 접두사가 붙는 추가적인 이름을 정의할수도 있는 향후의 파이썬 버전에 대한 사용자 코드의 이식성을 위태롭게 합니다.

헤더 파일들은 일반적으로 파이썬과 함께 설치됩니다. 유닉스에서는 헤더 파일들은 *prefix*/include/pythonversion/ 와 *exec_prefix*/include/pythonversion/ 안에 들어있습니다. prefix 와 exec_prefix 는 파이썬의 **configure** 스크립트에 해당하는 파라미터로 정의되며 버전 은 '%d.%d' % sys.version_info[:2] 입니다. 윈도우에서는 헤더 파일들은 *prefix*/include 안에 설치되어 있습니다. prefix 는 인스톨러에 의해 지정된 설치 디렉터리 입니다.

헤더를 인클루드 하기 위해선 두 디렉터리 모두 (두 디렉터리가 다를 경우) 컴파일러의 검색 패스에 추가하십시오. 부모 디렉터리를 검색 패스에 추가해서 #include <pythonX.Y/Python.h> 처럼 사용해서는 안됩니다. prefix 아래의 플랫폼 독립적인 헤더들이 exec_prefix 에서 플랫폼 종속적인 헤더를 인클루드하기 때문에 멀티플랫폼 빌드가 고장날 것입니다.

C++ 사용자들은 파이썬/C API 가 C 만을 사용하여 정의되었어도 헤더 파일들이 extern "C" 로 진입점을 제대로 선언한다는 점에 유의해야 합니다. C++ 에서 파이썬/C API 를 사용하기 위해 특별한 조치를 취할 필요는 없습니다.

1.4 유용한 매크로들

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (for example, *Py_RETURN_NONE*, *PyMODINIT_FUNC*). Others of a more general utility are defined here. This is not necessarily a complete listing.

Py_ABS(x)

x 의 절댓값을 반환합니다.

Added in version 3.3.

Py_ALWAYS_INLINE

Ask the compiler to always inline a static inline function. The compiler can ignore it and decide to not inline the function.

It can be used to inline performance critical static inline functions when building Python in debug mode with function inlining disabled. For example, MSC disables function inlining when building in debug mode.

Marking blindly a static inline function with Py_ALWAYS_INLINE can result in worse performances (due to increased code size for example). The compiler is usually smarter than the developer for the cost/benefit analysis.

If Python is built in debug mode (if the *Py_DEBUG* macro is defined), the *Py_ALWAYS_INLINE* macro does nothing.

It must be specified before the function return type. Usage:

```
static inline Py_ALWAYS_INLINE int random(void) { return 4; }
```

Added in version 3.11.

Py_CHARMASK(c)

인자는 문자 또는 [-128, 127] 나 [0, 255] 사이의 정수여야 합니다. 이 매크로는 `unsigned char` 로 캐스팅된 `c` 를 반환합니다

Py_DEPRECATED(version)

폐지(deprecated) 선언에 사용하십시오. 이 매크로는 심볼 이름 앞에 위치해야 합니다

예제:

```
Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
```

버전 3.8에서 변경: MSVC 지원을 추가했습니다.

Py_GETENV(s)

`getenv(s)` 와 유사하지만 `-E` 가 커맨드라인으로 전달된 경우 `NULL` 을 반환합니다 (*PyConfig.use_environment*를 참조하세요)

Py_MAX(x, y)

`x` 와 `y` 사이의 최댓값을 반환합니다.

Added in version 3.3.

Py_MEMBER_SIZE(type, member)

(`type`) 구조체의 `member` 의 크기를 바이트로 반환합니다.

Added in version 3.6.

Py_MIN(x, y)

`x` 와 `y` 사이의 최솟값을 반환합니다.

Added in version 3.3.

Py_NO_INLINE

Disable inlining on a function. For example, it reduces the C stack consumption: useful on LTO+PGO builds which heavily inline code (see bpo-33720).

사용법:

```
Py_NO_INLINE static int random(void) { return 4; }
```

Added in version 3.11.

Py_STRINGIFY(x)

`x` 를 C 문자열로 변환합니다. 예를 들어 `Py_STRINGIFY(123)` 은 `"123"` 을 반환합니다.

Added in version 3.4.

Py_UNREACHABLE()

의도적으로 도달할 수 없는 코드 경로가 있을 경우에 이 매크로를 사용하십시오. 예를 들어, `switch` 문에서 가능한 모든 값이 `case` 절에서 다뤄지는 경우에 `default:` 절에서 사용할 수 있습니다. `assert(0)` 또는 `abort()` 대신 사용하십시오.

릴리즈 모드에서 이 매크로는 컴파일러가 코드를 최적화하는데 도움이 되며 도달할 수 없는 코드에 대한 경고를 방지합니다. 예를 들어, 이 매크로는 릴리즈 모드에서 GCC의 `__builtin_unreachable()` 로 구현됩니다.

`Py_UNREACHABLE()` 의 용도는 반환하지 않지만 `_Py_NO_RETURN` 을 선언하지 않은 함수를 호출하는 것입니다

코드 경로가 매우 가능성이 낮지만 예외적인 경우에 도달할 수 있는 경우, 이 매크로를 사용해서는 안됩니다. 예를 들어, 메모리가 부족하거나 시스템 콜이 예상 범위를 벗어나는 값을 반환했을 경우에는 호출자에게 에러를 보고하는 것이 좋습니다. 호출자에게 에러를 보고할 수 없는 경우 *Py_FatalError()* 를 사용할 수 있습니다.

Added in version 3.7.

Py_UNUSED(arg)

함수의 미사용 인자에 사용하여 컴파일러 경고를 무시합니다. 예시: `int func(int a, int Py_UNUSED(b)) { return a; }`.

Added in version 3.4.

PyDoc_STRVAR(name, str)

독스트링에서 사용 가능한 `name` 이란 이름의 변수를 생성합니다. 파이썬이 독스트링 없이 빌드되었다면 변수의 값은 비어있을 것입니다.

PEP 7 에 명시된 것처럼 파이썬을 독스트링 없이 빌드하기 위해 *PyDoc_STRVAR* 를 독스트링에 사용하십시오

예제:

```
PyDoc_STRVAR(pop_doc, "Remove and return the rightmost element.");

static PyMethodDef deque_methods[] = {
    // ...
    {"pop", (PyCFunction)deque_pop, METH_NOARGS, pop_doc},
    // ...
}
```

PyDoc_STR(str)

주어진 문자열에 대한 독스트링을 생성합니다. 독스트링이 비활성화 되어있을 경우엔 빈 문자열을 생성합니다.

PEP 7 에 명시된 것처럼 독스트링 없이 파이썬을 빌드할 수 있도록 독스트링을 명시할 때 *PyDoc_STR* 을 사용하십시오.

예제:

```
static PyMethodDef pysqlite_row_methods[] = {
    {"keys", (PyCFunction)pysqlite_row_keys, METH_NOARGS,
        PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.5 객체, 형 그리고 참조 횟수

대부분의 파이썬/C API 함수는 하나 이상의 인자와 *PyObject** 형의 반환 값을 가집니다. 해당 형은 임의의 파이썬 객체를 나타내는 오파크(Opaque) 데이터 형에 대한 포인터입니다. 할당, 스코프 규칙, 인자 전달 등 대부분의 상황에서 파이썬 언어가 모든 파이썬 객체 형을 동일한 방식으로 취급하기 때문에 단일한 C 형으로 표현하는 것이 적절합니다. 대부분의 객체는 힙에 존재합니다. *PyObject* 형의 자동 변수나 정적 변수를 선언해서는 안되며 *PyObject** 형의 포인터 변수만을 선언할 수 있습니다. 유일한 예외는 형 객체입니다. 형 객체들은 절대 할당이 해제돼서는 안되기 때문에 형 객체들은 보통 정적 *PyTypeObject* 객체들입니다.

모든 파이썬 객체는 (파이썬 정수조차도) 형(*type*) 과 참조 횟수(*reference count*) 를 가지고 있습니다. 객체의 형은 객체의 종류를 결정합니다. (예를 들어 정수, 리스트, 또는 사용자 정의 함수 등. types 에 추가적인 형들에 대해 설명되어 있습니다.) 잘 알려진 형에는 객체가 해당 형인지를 확인하는 매크로가 있습니다. 예를 들어 `PyList_Check(a)` 는 *a* 가 가리키는 객체가 파이썬 리스트일 경우에만 참입니다.

1.5.1 참조 횟수

오늘날의 컴퓨터는 유한한(그리고 자주 제한되는) 양의 메모리를 가지고 있기 때문에 참조 횟수는 매우 중요합니다; 참조 횟수는 객체에 대해 강한 참조를 갖는 위치의 갯수를 셉니다. 객체를 참조하는 위치는 다른 객체일 수도 있고, C 전역 변수이거나 C 함수 안의 지역 변수일 수도 있습니다. 객체에 대한 마지막 강한 참조가 해제될 때 (즉, 객체의 참조 횟수가 0이 되면), 객체는 할당이 해제됩니다. 객체가 다른 객체에 대한 참조를 가지고 있을 경우, 그 참조들도 해제됩니다. 이들에 대한 참조가 더는 없으면, 다른 객체들이 차례로 할당이 해제될 수 있습니다. (서로를 참조하는 객체에 대해 분명한 문제점이 있으나 현재 해결책은 "그러지 마세요" 입니다.)

참조 횟수는 항상 명시적으로 조작됩니다. 일반적인 방법은 매크로 *Py_INCREF()* 를 사용하여 객체에 대한 새로운 참조를 취하고(즉, 객체의 참조 횟수를 하나 증가시키고), *Py_DECREF()* 를 사용하여 참조를 해제하는 (즉, 참조 횟수를 하나 감소시키는) 것입니다. *Py_DECREF()* 매크로는 참조 횟수가 0이 되는지를 확인하고 객체의 할당해제자(deallocator) 를 호출해야 하므로 참조 횟수를 증가시키는 매크로보다 상당히 복잡합니다. 할당 해제자는 객체의 형 구조체에 포함된 함수 포인터입니다. 형별 할당 해제자는 리스트와 같은 복합 객체 형에 대해서 객체에 포함된 객체에 포함 된 다른 객체에 대한 참조를 해제하고 필요한 추가 마무리 작업을 수행합니다. 참조 횟수가 오버플로우 될 가능성은 없습니다. 참조 횟수를 저장하기 위해서 가상 메모리 안에서 구별되는 메모리 위치 만큼 (`sizeof(Py_ssize_t) >= sizeof(void*)` 라고 가정할 경우) 많은 비트가 사용됩니다. 따라서 참조 횟수 증가는 간단한 연산입니다

객체에 대한 포인터를 포함하는 모든 지역 변수에 대해 객체에 대한 강한 참조를 잡을 (즉, 참조 횟수를 증가시킬) 필요는 없습니다. 이론적으로는 객체의 참조 횟수는 변수가 객체를 가리킬 때 증가하고 변수가 스코프를 벗어날 때 감소합니다. 하지만 이 두 동작은 서로를 상쇄시키기 때문에 결과적으로 참조 횟수는 변하지 않습니다. 참조 횟수를 사용해야만 하는 이유는 우리의 변수가 객체를 참조하는 한 객체가 할당 해제되지 않도록 하기 위함입니다. 만약 객체에 대한 적어도 하나 이상의 다른 참조가 우리의 변수만큼 오래 살아있을 것을 안다면 일시적으로 새로운 *strong reference*를 취할 (즉, 참조 횟수를 증가시킬) 필요가 없습니다. 이런 현상이 발생하는 중요한 상황은 파이썬이 호출하는 확장 모듈의 C 함수에 인자로 넘겨지는 객체에서 발생합니다. 호출 메커니즘은 호출하는동안 모든 인자에 대한 참조를 보유하도록 보장합니다.

그러나 흔히 하기 쉬운 실수는 리스트에서 객체를 가져와 새 참조를 취하지 않고 유지하는 것입니다. 다른 연산이 리스트에서 객체를 제거해 그 참조를 해제하면 객체가 할당 해제될수도 있습니다. 진짜 위험은 무해해 보이는 연산이 이를 실행할 수도 있는 임의의 파이썬 코드를 실행할수도 있다는 것입니다. *Py_DECREF()* 에서 사용자로 컨트롤을 돌아가게 하는 코드 경로가 있기 때문에 사실 거의 모든 연산이 잠재적으로 위험합니다.

안전한 접근방식은 제네릭 연산들 (`PyObject_`, `PyNumber_`, `PySequence_` 또는 `PyMapping_` 로 시작하는 이름을 가진 함수들) 을 사용하는 것입니다. 이러한 연산들은 항상 반환하는 모든 객체에 대해 새로운 강한 참조를 만듭니다 (즉, 참조 횟수를 증가시킵니다). 이는 함수 호출자에게 연산이 반환된 객체와의 작업이 끝났을 경우 *Py_DECREF()* 를 호출할 책임을 지웁니다. 이건 곧 습관화 될 것입니다.

참조 횟수 상세

파이썬/C API 에서 함수의 참조 횟수 동작은 참조의 소유권 으로 가장 잘 설명됩니다. 소유권은 객체가 아닌 참조에 관련되어 있습니다. (객체는 소유되지 않습니다: 객체는 항상 공유됩니다.) "참조를 소유하는 것" 은 더 이상 참조가 필요하지 않을 때 Py_DECREF 를 호출할 책임이 있다는 것을 의미합니다. 소유권은 양도될 수도 있습니다. 즉, 참조의 소유권을 받은 코드가 소유권이 더 이상 필요하지 않을 때 *Py_DECREF()* 또는 *Py_XDECREF()* 를 호출하거나 책임을 (대게 자신의 호출자에게) 넘김으로써 결국 참조를 해제할 책임이 생긴다는 것을 의미합니다. 함수가 참조의 소유권을 호출자에게 넘길때 호출자는 새로운 참조를 받는다고 합니다. 소유권이 옮겨지지 않을 때 호출자는 참조를 빌린다 라고 합니다. 빌린 참조를 위해서는 아무것도 할 필요가 없습니다.

반대로 호출 함수에게 객체에 대한 참조를 넘길 때는 두가지 가능성이 있습니다: 함수는 객체에 대한 참조를 훔칠 수도, 그러지 않을 수도 있습니다. 참조를 훔치는 것은 함수에 참조를 전달할 때 해당 함수가 전달된 참조를 소유한다고 가정하고 더 이상 책임을 지지 않는다는 것을 의미합니다.

참조를 훔치는 함수는 거의 없습니다. 주목할만한 두가지 예외는 *PyList_SetItem()* 과 *PyTuple_SetItem()* 입니다. 이 두가지 함수는 요소에 대한 참조를 훔칩니다(단, 요소를 넣을 튜플이나 리스트에 대한 참조는 훔치지 않습니다.). 이 함수들은 새로 만들어진 객체들로 튜플이나 리스트를 채우는 일반적인 관행 때문에 참조를 훔치도록 설계되었습니다. 예를 들어, 튜플을 만드는 코드 (`1, 2, "three"`) 는 다음과 같을 수 있습니다. (잠시 에러 처리는 잊어버리십시오. 더 좋은 방법으로 코딩하는 방법은 아래에 나와 있습니다.):

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

여기서 *PyTuple_SetItem()* 는 *PyLong_FromLong()* 가 반환한 참조를 곧바로 훔칩니다. 객체에 대한 참조가 훔쳐져도 계속 객체를 사용하려면 참조를 훔치는 함수를 호출하기 전에 *Py_INCREF()* 를 다른 참조를 가져오는데 사용하십시오.

덧붙이자면, *PyTuple_SetItem()* 은 튜플에 요소를 넣는 유일한 방법입니다. 튜플은 불변 자료형이기 때문에 *PySequence_SetItem()* 과 *PyObject_SetItem()* 는 튜플에 요소를 넣는 것을 거부합니다. *PyTuple_SetItem()* 은 직접 만들고 있는 튜플에만 사용되어야 합니다.

리스트를 채우는 동일한 의미의 코드는 *PyList_New()* 와 *PyList_SetItem()* 을 사용해 만들 수 있습니다.

하지만 실제로는 이렇게 튜플 또는 리스트를 만들고 채우는 경우는 드뭅니다. 일반적인 객체들을 형식 문자열(*format string*)로 지시되는 C 값으로부터 만들어낼 수 있는 제네릭 함수 *Py_BuildValue()* 가 있습니다. 예를 들어, 위의 두 블록의 코드를 다음 코드로 대체할 수 있습니다. (에러 검사도 처리합니다.):

```
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

PyObject_SetItem() 과 그 친구들은 함수에 전달되는 인자처럼 참조만 빌리고 있는 요소와 함께 사용하는 것이 일반적입니다. 이 경우 새로운 참조를 취하지 않고 참조를 제공할 수 있으므로 참조에 대한 함수들의 동작이 훨씬 멀쩡합니다:

```
int
set_all(PyObject *target, PyObject *item)
{
    Py_ssize_t i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

함수 반환 값에 대해서는 상황이 약간 다릅니다. 대부분의 함수에 참조를 전달해도 해당 참조에 대한 소유권 책임이 바뀌진 않지만 객체에 대한 참조를 제공하는 많은 함수는 참조의 소유권을 제공합니다. 이유는 간단합니다. 대부분의 경우에서 반환된 객체는 즉석에서 생성되고 반환된 참조는 객체에 대한 유일한 참조입니다. 따라서 *PyObject_GetItem()* 과 *PySequence_GetItem()* 처럼 객체에 대한 참조를 반환하는 제네릭 함수들은 언제나 새로운 참조를 반환합니다 (호출자가 객체의 소유자가 됩니다).

함수의 의해 반환된 함수를 소유하고 있는지는 어떤 함수를 호출하느냐에 따라 달라진다는 것을 아는 것이 중요합니다. — 깃털 (함수에 인자로 전달된 객체의 형) 은 해당되지 않습니다! 따라서 *PyList_GetItem()* 를 사용하여 리스트에서 항목을 가져오면 참조를 소유하지 않습니다. — 하지만 동일한 인자를 받는

PySequence_GetItem() 를 사용하여 리스트에서 항목을 가져온다면 반환된 객체에 대한 참조를 소유하게 됩니다.

다음은 정수 리스트에 있는 항목의 합계를 구하는 함수를 작성하는 방법의 예시입니다. 한 번은 *PyList_GetItem()* 를 사용하고, 한 번은 *PySequence_GetItem()* 을 사용합니다.

```
long
sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyLong_Check(item)) continue; /* Skip non-integers */
        value = PyLong_AsLong(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long, bail out */
            return -1;
        total += value;
    }
    return total;
}
```

```
long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* Not a sequence, or other failure */
        if (PyLong_Check(item)) {
            value = PyLong_AsLong(item);
            Py_DECREF(item);
            if (value == -1 && PyErr_Occurred())
                /* Integer too big to fit in a C long, bail out */
                return -1;
            total += value;
        }
        else {
            Py_DECREF(item); /* Discard reference ownership */
        }
    }
    return total;
}
```

1.5.2 형

파이썬/C API에서 중요한 역할을 하는 다른 데이터 형은 거의 없습니다. 대부분은 `int`, `long`, `double` 그리고 `char*` 같은 평범한 C 형입니다. 모듈에서 내보내는 함수나 새 객체 형의 데이터 속성들을 나열하는데 사용되는 정적 테이블을 표현하는데 사용되는 구조체 형이 몇가지 있으며, 복소수를 표현하기 위해 사용되는 구조체 형도 있습니다. 이러한 형들은 해당 형들을 사용하는 함수와 함께 다뤄질 것입니다.

type **Py_ssize_t**

Part of the 안정 ABI. A signed integral type such that `sizeof(Py_ssize_t) == sizeof(size_t)`. C99 doesn' t define such a thing directly (size_t is an unsigned integral type). See **PEP 353** for details. `PY_SSIZE_T_MAX` is the largest positive value of type *Py_ssize_t*.

1.6 예외

파이썬 프로그래머는 특정한 에러 처리가 필요할 경우에만 에러를 처리하면 됩니다. 처리되지 않은 예외는 사용자에게 전달되는 최상위 인터프리터까지 스택 트레이스백과 함께 자동으로 호출자, 호출자의 호출자 등으로 전파됩니다.

그러나 C 프로그래머들에게 에러 검사는 항상 명시적이어야만 합니다. 파이썬/C API의 모든 함수는 해당 함수의 문서에서 명시하지 않는 한 예외를 발생시킬 수 있습니다. 일반적으로 함수에 에러가 발생하면 함수는 예외를 설정하고 소유하고 있는 모든 객체에 대한 참조를 취소하고 에러 표시기를 반환합니다. 달리 문서화되지 않은 경우 표시기는 함수의 반환 형에 따라 `NULL` 또는 `-1` 입니다. 일부 함수는 에러를 의미하는 거짓과 함께 참/거짓의 불리언 결과를 반환합니다. 아주 일부의 함수는 명시적인 에러 표시기가 없거나 모호한 반환값을 가지며 *PyErr_Occurred()* 를 사용하여 명시적인 점검을 요구합니다. 이런 예외는 항상 명시적으로 문서화됩니다.

예외 상태는 스레드 별 공간에서 관리됩니다. (스레드를 사용하지 않는 프로그램에서는 전역 공간을 사용한다는 말과 같습니다.) 스레드는 예외가 발생했거나, 발생하지 않았거나의 두가지 상태 중 하나일 수 있습니다. 함수 *PyErr_Occurred()* 는 이 상태를 확인하기 위해 사용할 수 있습니다. 해당 함수는 예외가 발생했을 경우 예외 형 객체에 대한 빌린 참조를 반환합니다. 예외가 발생하지 않았을 경우엔 `NULL` 을 반환합니다. 예외 상태를 설정하기 위한 여러가지 함수들이 있습니다: *PyErr_SetString()* 는 예외 상태를 설정하기 위해 가장 보편적인 (가장 일반적인 것은 아니지만) 함수입니다. *PyErr_Clear()* 는 예외 상태를 지웁니다.

전체 예외 상태는 예외 형, 해당 예외 값, 트레이스백이라는 세가지 객체로 구성됩니다. (셋 모두 `NULL` 일 수 있습니다.) 이 세가지 객체는 파이썬의 `sys.exc_info()` 의 결과와 같은 의미를 가지고 있지만 동일하지는 않습니다. 파이썬 객체는 `try` … `except` 문으로 처리되는 마지막 예외를 표현하는 반면 C 수준 예외는 `sys.exc_info()` 와 그 친구들로 예외를 전송하는 파이썬 바이트코드 인터프리터의 메인 루프에 도달할 때까지 C 함수들 간에 전달되는 동안에만 존재합니다.

파이썬 1.5부터 선호되어 온 파이썬 코드에서의 스레드 안전한 예외 상태 접근 방법은 파이썬 코드를 위해 스레드 별 예외 상태를 반환하는 `sys.exc_info()` 함수를 호출하는 것입니다. 또한 예외 상태에 접근하는 양쪽 방법의 의미도 바뀌어 에러를 포착하는 함수가 호출자의 예외 상태를 보존하기 위해 스레드의 예외를 저장하고 복원합니다. 이는 평범해 보이는 함수가 처리중인 예외를 덮어씌우는 것으로 인한 예외 처리 코드의 흔한 버그를 방지합니다. 또한 트레이스백의 스택 프레임에 의하여 참조되는 객체들에 대해 종종 원하지 않은 수명 증가가 일어나는 것을 방지합니다

일반적으로 어떤 작업을 수행하기 위해 다른 함수를 호출하는 함수는 호출된 함수가 예외를 일으켰는지 확인해야만 하며 만약 예외가 일어났다면 호출자에게 예외 상태를 전달해야 합니다. 소유하고 있는 모든 객체에 대한 참조를 버리고 에러 표시기를 반환해야 하지만 다른 예외를 설정해서는 안됩니다. — 방금 일어난 예외를 덮어씌우고 정확한 에러 원인에 대한 중요한 정보를 잃어버리게 됩니다.

예외를 감지하고 전달하는 간단한 예가 아래 `sum_sequence()` 예시에 나와 있습니다. 우연히 이 예시에선 에러를 감지했을 때 소유하고 있는 참조를 정리할 필요가 없습니다. 그 다음 예시 함수에서는 몇가지 에러 정리작업을 보여줍니다. 먼저 파이썬을 좋아하는 이유를 상기시키기 위해 같은 의미의 파이썬 코드를 제공합니다:

```
def incr_item(dict, key):
    try:
        item = dict[key]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    except KeyError:
        item = 0
    dict[key] = item + 1
```

다음은 같은 의미의 웅장한 C 코드입니다:

```
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;

    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

 error:
    /* Cleanup code, shared by success and failure path */

    /* Use Py_XDECREF() to ignore NULL references */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 for error, 0 for success */
}
```

이 예시는 C `goto` 문의 허용된 사용방법을 보여줍니다! 이 예시는 특정한 예외를 처리하기 위한 *PyErr_ExceptionMatches()* 와 *PyErr_Clear()* 의 사용 방법과 *Py_XDECREF()* 를 사용하여 소유하고 있는 `NULL` 일 수도 있는 참조를 삭제하는 방법을 표현합니다. (이름에 있는 `'X'` 를 주목하십시오. *Py_DECREF()* 는 `NULL` 참조와 마주치면 충돌을 일으킵니다.) 이 예시를 수행하려면 소유하고 있는 참조를 보유하는데 사용하는 변수를 `NULL` 로 초기화하는 것이 중요합니다. 마찬가지로 반환 값은 `-1` (실패) 로 설정되고 마지만 호출이 성공한 뒤에야 성공으로 설정됩니다.

1.7 파이썬 임베딩하기

확장 작성자들과는 달리 파이썬 인터프리터를 임베딩 하는 사람들만이 걱정해야 하는 한가지 중요한 문제는 파이썬 인터프리터의 초기화, 그리고 아마도 마무리일 것입니다. 인터프리터의 대부분의 기능은 인터프리터가 초기화 된 이후에 사용할 수 있습니다.

기본적인 초기화 함수는 *Py_Initialize()* 입니다. 이 함수는 로드된 모듈 테이블을 초기화 하고 기본 모듈인 `builtins`, `__main__`, 그리고 `sys` 를 생성합니다. 또한 모듈 검색 경로 (`sys.path`) 를 초기화합니다

Py_Initialize() 는 "스크립트 인자 리스트" (`sys.argv`) 를 설정하지 않습니다. 만약 후에 실행될 파이썬 코드가 이 변수를 필요로 한다면, 설정 *PyConfig.argv*와 *PyConfig.parse_argv*는 반드시 설정되어야 합니다: 파이썬 초기화 구성을 참조하세요.

대부분의 시스템에서 (특별히 유닉스와 윈도우는 세부적인 부분이 조금 다르긴 하지만) *Py_Initialize()* 는 파이썬 인터프리터를 기준으로 고정된 위치에 파이썬 라이브러리가 있다고 가정하여 표준 파이썬 인터프리터 실행 파일에 대한 최선의 추측을 바탕으로 바탕으로 모듈 검색 경로를 계산합니다. 특히 셸 명령어 검색 경로 (환경 변수 `PATH`) 에서 `python` 이라는 이름의 실행 파일이 발견되는 부모 디렉터리를 기준으로 `lib/python`*X.Y* 같은 이름을 가진 디렉터리를 찾습니다.

예를 들어 파이썬 실행 파일이 `/usr/local/bin/python` 에서 발견된다면 라이브러리는 `/usr/local/lib/python`*X.Y* 에 있는 것으로 가정합니다. (실제로 이 특정 경로는 `PATH` 를 따라 `python` 이라는 이름의 실행 파일이 발견되지 않을 때 사용되는 "fallback" 경로이기도 합니다.) 유저는 환경 변수 `PYTHONHOME` 를 설정하여 이 동작을 재정의하거나 `PYTHONPATH` 를 설정하여 표준 경로 앞에 추가적인 디렉터리를 추가할 수 있습니다.

파이썬을 임베딩하는 애플리케이션은 *Py_InitializeFromConfig()* 를 호출하기 전에 *PyConfig.program_name* 을 설정하여 검색을 제어할 수 있습니다. `PYTHONHOME` 는 여전히 이 값을 재정의하며 `PYTHONPATH` 는 여전히 표준 경로 앞에 추가된다는 점에 유의하십시오. 완전한 제어가 필요한 애플리케이션은 *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, 그리고 *Py_GetProgramFullPath()* 의 자체적인 구현을 제공할 필요가 있습니다. (모두 `Modules/getpath.c` 에 정의되어 있습니다.)

가끔은 파이썬을 "uninitialize" 하는 것이 바람직합니다. 예를 들어 애플리케이션이 다시 시작하거나 (*Py_Initialize()* 다시 호출하기) 애플리케이션에서 파이썬의 사용이 끝나 파이썬이 할당한 메모리를 해제하려고 할 수 있습니다. *Py_FinalizeEx()* 를 호출하여 이를 달성할 수 있습니다. 함수 *Py_IsInitialized()* 는 파이썬이 현재 초기화된 상태에 있을 경우 참을 반환합니다. 이 함수들에 대한 자세한 내용은 다른 장에서 제공됩니다. *Py_FinalizeEx()* 가 파이썬 인터프리터가 할당한 모든 메모리를 해제하지는 않는다는 점에 유의해야 합니다. 예를 들어, 현재 확장 모듈에서 할당한 메모리는 해제할 수 없습니다.

1.8 디버깅 빌드

파이썬은 인터프리터와 확장 모듈들에 대한 추가적인 검사를 가능하게 하는 여러 매크로를 사용하여 빌드될 수 있습니다. 이러한 검사는 런타임에 많은 오버헤드를 추가하는 경향이 있으므로 기본적으로 실행되지 않습니다.

다양한 유형의 디버깅 빌드의 전체 목록은 파이썬 소스 배포판 안의 `Misc/SpecialBuilds.txt` 파일에 있습니다. 참조 횟수 추적, 메모리 할당자 디버깅, 메인 인터프리터 루프의 저수준 프로파일링을 지원하는 빌드들을 사용할 수 있습니다. 이 섹션에서는 가장 자주 사용되는 빌드만 설명합니다.

Py_DEBUG

`Py_DEBUG` 매크로가 정의된 인터프리터를 컴파일하면 일반적으로 파이썬의 디버그 빌드가 일반적으로 의미하는 빌드가 생성됩니다. `Py_DEBUG` 는 `./configure` 명령에 `--with-pydebug` 를 추가하여 유닉스 빌드에서 활성화됩니다. 또한 파이썬 전용이 아닌 `_DEBUG` 매크로의 존재를 암시합니다. 유닉스 빌드에서 `Py_DEBUG` 가 활성화 되어있으면 컴파일러 최적화가 비활성화됩니다.

아래에 설명된 참조 횟수 디버깅 외에도 추가적인 검사가 수행됩니다, 파이썬 디버그 빌드를 참조하세요.

`Py_TRACE_REFS` 를 정의하면 참조 추적이 활성화됩니다 (`configure --with-trace-refs option`를 참조하세요). 정의된 경우 모든 *PyObject* 에 두 개의 추가 필드를 추가함으로써 활성 객체에 대한 이중 원형 연결 리스트가 유지됩니다. 또한 모든 할당이 추적됩니다. 인터프리터 종료시에는 현재 사용되고 있는 모든 참조가 출력됩니다. (대화식 모드에서는 인터프리터가 명령문을 실행할 때마다 이후에 출력됩니다.)

자세한 내용은 파이썬 소스 배포판 안의 `Misc/SpecialBuilds.txt` 를 참조하십시오.

1.9 Recommended third party tools

The following third party tools offer both simpler and more sophisticated approaches to creating C, C++ and Rust extensions for Python:

- Cython
- cffi
- HPy
- nanobind (C++)
- Numba
- pybind11 (C++)
- PyO3 (Rust)
- SWIG

Using tools such as these can help avoid writing code that is tightly bound to a particular version of CPython, avoid reference counting errors, and focus more on your own code than on using the CPython API. In general, new versions of Python can be supported by updating the tool, and your code will often use newer and more efficient APIs automatically. Some tools also support compiling for other implementations of Python from a single set of sources.

These projects are not supported by the same people who maintain Python, and issues need to be raised with the projects directly. Remember to check that the project is still maintained and supported, as the list above may become outdated.

더 보기

Python Packaging User Guide: Binary Extensions
The Python Packaging User Guide not only covers several available tools that simplify the creation of binary extensions, but also discusses the various reasons why creating an extension module may be desirable in the first place.

CHAPTER 2

C API Stability

Unless documented otherwise, Python' s C API is covered by the Backwards Compatibility Policy, **PEP 387**. Most changes to it are source-compatible (typically by only adding new API). Changing existing API or removing API is only done after a deprecation period or to fix serious issues.

CPython' s Application Binary Interface (ABI) is forward- and backwards-compatible across a minor release (if these are compiled the same way; see *Platform Considerations* below). So, code compiled for Python 3.10.0 will work on 3.10.8 and vice versa, but will need to be compiled separately for 3.9.x and 3.11.x.

There are two tiers of C API with different stability expectations:

- *Unstable API*, may change in minor versions without a deprecation period. It is marked by the `PyUnstable` prefix in names.
- *Limited API*, is compatible across several minor releases. When `Py_LIMITED_API` is defined, only this subset is exposed from `Python.h`.

These are discussed in more detail below.

Names prefixed by an underscore, such as `_Py_InternalState`, are private API that can change without notice even in patch releases. If you need to use this API, consider reaching out to CPython developers to discuss adding public API for your use case.

2.1 Unstable C API

Any API named with the `PyUnstable` prefix exposes CPython implementation details, and may change in every minor release (e.g. from 3.9 to 3.10) without any deprecation warnings. However, it will not change in a bugfix release (e.g. from 3.10.0 to 3.10.1).

It is generally intended for specialized, low-level tools like debuggers.

Projects that use this API are expected to follow CPython development and spend extra effort adjusting to changes.

2.2 안정적인 응용 프로그램 바이너리 인터페이스

For simplicity, this document talks about *extensions*, but the Limited API and Stable ABI work the same way for all uses of the API – for example, embedding Python.

2.2.1 Limited C API

Python 3.2 introduced the *Limited API*, a subset of Python' s C API. Extensions that only use the Limited API can be compiled once and be loaded on multiple versions of Python. Contents of the Limited API are *listed below*.

Py_LIMITED_API

Define this macro before including `Python.h` to opt in to only use the Limited API, and to select the Limited API version.

Define `Py_LIMITED_API` to the value of *PY_VERSION_HEX* corresponding to the lowest Python version your extension supports. The extension will be ABI-compatible with all Python 3 releases from the specified one onward, and can use Limited API introduced up to that version.

Rather than using the `PY_VERSION_HEX` macro directly, hardcode a minimum minor version (e.g. `0x030A0000` for Python 3.10) for stability when compiling with future Python versions.

You can also define `Py_LIMITED_API` to `3`. This works the same as `0x03020000` (Python 3.2, the version that introduced Limited API).

2.2.2 Stable ABI

To enable this, Python provides a *Stable ABI*: a set of symbols that will remain ABI-compatible across Python 3.x versions.

> **참고**
>
> The Stable ABI prevents ABI issues, like linker errors due to missing symbols or data corruption due to changes in structure layouts or function signatures. However, other changes in Python can change the *behavior* of extensions. See Python' s Backwards Compatibility Policy (**PEP 387**) for details.

The Stable ABI contains symbols exposed in the *Limited API*, but also other ones – for example, functions necessary to support older versions of the Limited API.

On Windows, extensions that use the Stable ABI should be linked against `python3.dll` rather than a version-specific library such as `python39.dll`.

On some platforms, Python will look for and load shared library files named with the `abi3` tag (e.g. `mymodule.abi3.so`). It does not check if such extensions conform to a Stable ABI. The user (or their packaging tools) need to ensure that, for example, extensions built with the 3.10+ Limited API are not installed for lower versions of Python.

All functions in the Stable ABI are present as functions in Python' s shared library, not solely as macros. This makes them usable from languages that don' t use the C preprocessor.

2.2.3 Limited API Scope and Performance

The goal for the Limited API is to allow everything that is possible with the full C API, but possibly with a performance penalty.

For example, while *PyList_GetItem()* is available, its "unsafe" macro variant *PyList_GET_ITEM()* is not. The macro can be faster because it can rely on version-specific implementation details of the list object.

Without `Py_LIMITED_API` defined, some C API functions are inlined or replaced by macros. Defining `Py_LIMITED_API` disables this inlining, allowing stability as Python' s data structures are improved, but possibly reducing performance.

By leaving out the `Py_LIMITED_API` definition, it is possible to compile a Limited API extension with a version-specific ABI. This can improve performance for that Python version, but will limit compatibility. Compiling with `Py_LIMITED_API` will then yield an extension that can be distributed where a version-specific one is not available – for example, for prereleases of an upcoming Python version.

2.2.4 Limited API Caveats

Note that compiling with `Py_LIMITED_API` is *not* a complete guarantee that code conforms to the *Limited API* or the *Stable ABI*. `Py_LIMITED_API` only covers definitions, but an API also includes other issues, such as expected semantics.

One issue that `Py_LIMITED_API` does not guard against is calling a function with arguments that are invalid in a lower Python version. For example, consider a function that starts accepting `NULL` for an argument. In Python 3.9, `NULL` now selects a default behavior, but in Python 3.8, the argument will be used directly, causing a `NULL` dereference and crash. A similar argument works for fields of structs.

Another issue is that some struct fields are currently not hidden when `Py_LIMITED_API` is defined, even though they' re part of the Limited API.

For these reasons, we recommend testing an extension with *all* minor Python versions it supports, and preferably to build with the *lowest* such version.

We also recommend reviewing documentation of all used API to check if it is explicitly part of the Limited API. Even with `Py_LIMITED_API` defined, a few private declarations are exposed for technical reasons (or even unintentionally, as bugs).

Also note that the Limited API is not necessarily stable: compiling with `Py_LIMITED_API` with Python 3.8 means that the extension will run with Python 3.12, but it will not necessarily *compile* with Python 3.12. In particular, parts of the Limited API may be deprecated and removed, provided that the Stable ABI stays stable.

2.3 Platform Considerations

ABI stability depends not only on Python, but also on the compiler used, lower-level libraries and compiler options. For the purposes of the *Stable ABI*, these details define a "platform". They usually depend on the OS type and processor architecture

It is the responsibility of each particular distributor of Python to ensure that all Python versions on a particular platform are built in a way that does not break the Stable ABI. This is the case with Windows and macOS releases from `python.org` and many third-party distributors.

2.4 Contents of Limited API

Currently, the *Limited API* includes the following items:

- *PY_VECTORCALL_ARGUMENTS_OFFSET*
- *PyAIter_Check()*
- *PyArg_Parse()*
- *PyArg_ParseTuple()*
- *PyArg_ParseTupleAndKeywords()*
- *PyArg_UnpackTuple()*
- *PyArg_VaParse()*
- *PyArg_VaParseTupleAndKeywords()*
- *PyArg_ValidateKeywordArguments()*
- *PyBaseObject_Type*
- *PyBool_FromLong()*
- *PyBool_Type*
- *PyBuffer_FillContiguousStrides()*

- *PyBuffer_FillInfo()*
- *PyBuffer_FromContiguous()*
- *PyBuffer_GetPointer()*
- *PyBuffer_IsContiguous()*
- *PyBuffer_Release()*
- *PyBuffer_SizeFromFormat()*
- *PyBuffer_ToContiguous()*
- PyByteArrayIter_Type
- *PyByteArray_AsString()*
- *PyByteArray_Concat()*
- *PyByteArray_FromObject()*
- *PyByteArray_FromStringAndSize()*
- *PyByteArray_Resize()*
- *PyByteArray_Size()*
- *PyByteArray_Type*
- PyBytesIter_Type
- *PyBytes_AsString()*
- *PyBytes_AsStringAndSize()*
- *PyBytes_Concat()*
- *PyBytes_ConcatAndDel()*
- PyBytes_DecodeEscape()
- *PyBytes_FromFormat()*
- *PyBytes_FromFormatV()*
- *PyBytes_FromObject()*
- *PyBytes_FromString()*
- *PyBytes_FromStringAndSize()*
- PyBytes_Repr()
- *PyBytes_Size()*
- *PyBytes_Type*
- *PyCFunction*
- *PyCFunctionFast*
- *PyCFunctionFastWithKeywords*
- *PyCFunctionWithKeywords*
- PyCFunction_GetFlags()
- PyCFunction_GetFunction()
- PyCFunction_GetSelf()
- *PyCFunction_New()*
- *PyCFunction_NewEx()*
- PyCFunction_Type

- *PyCMethod_New()*
- *PyCallIter_New()*
- *PyCallIter_Type*
- *PyCallable_Check()*
- *PyCapsule_Destructor*
- *PyCapsule_GetContext()*
- *PyCapsule_GetDestructor()*
- *PyCapsule_GetName()*
- *PyCapsule_GetPointer()*
- *PyCapsule_Import()*
- *PyCapsule_IsValid()*
- *PyCapsule_New()*
- *PyCapsule_SetContext()*
- *PyCapsule_SetDestructor()*
- *PyCapsule_SetName()*
- *PyCapsule_SetPointer()*
- PyCapsule_Type
- PyClassMethodDescr_Type
- *PyCodec_BackslashReplaceErrors()*
- *PyCodec_Decode()*
- *PyCodec_Decoder()*
- *PyCodec_Encode()*
- *PyCodec_Encoder()*
- *PyCodec_IgnoreErrors()*
- *PyCodec_IncrementalDecoder()*
- *PyCodec_IncrementalEncoder()*
- *PyCodec_KnownEncoding()*
- *PyCodec_LookupError()*
- *PyCodec_NameReplaceErrors()*
- *PyCodec_Register()*
- *PyCodec_RegisterError()*
- *PyCodec_ReplaceErrors()*
- *PyCodec_StreamReader()*
- *PyCodec_StreamWriter()*
- *PyCodec_StrictErrors()*
- *PyCodec_Unregister()*
- *PyCodec_XMLCharRefReplaceErrors()*
- *PyComplex_FromDoubles()*
- *PyComplex_ImagAsDouble()*

- *PyComplex_RealAsDouble()*
- *PyComplex_Type*
- *PyDescr_NewClassMethod()*
- *PyDescr_NewGetSet()*
- *PyDescr_NewMember()*
- *PyDescr_NewMethod()*
- PyDictItems_Type
- PyDictIterItem_Type
- PyDictIterKey_Type
- PyDictIterValue_Type
- PyDictKeys_Type
- *PyDictProxy_New()*
- PyDictProxy_Type
- PyDictRevIterItem_Type
- PyDictRevIterKey_Type
- PyDictRevIterValue_Type
- PyDictValues_Type
- *PyDict_Clear()*
- *PyDict_Contains()*
- *PyDict_Copy()*
- *PyDict_DelItem()*
- *PyDict_DelItemString()*
- *PyDict_GetItem()*
- *PyDict_GetItemRef()*
- *PyDict_GetItemString()*
- *PyDict_GetItemStringRef()*
- *PyDict_GetItemWithError()*
- *PyDict_Items()*
- *PyDict_Keys()*
- *PyDict_Merge()*
- *PyDict_MergeFromSeq2()*
- *PyDict_New()*
- *PyDict_Next()*
- *PyDict_SetItem()*
- *PyDict_SetItemString()*
- *PyDict_Size()*
- *PyDict_Type*
- *PyDict_Update()*
- *PyDict_Values()*

- *PyEllipsis_Type*
- PyEnum_Type
- *PyErr_BadArgument()*
- *PyErr_BadInternalCall()*
- *PyErr_CheckSignals()*
- *PyErr_Clear()*
- PyErr_Display()
- *PyErr_DisplayException()*
- *PyErr_ExceptionMatches()*
- *PyErr_Fetch()*
- *PyErr_Format()*
- *PyErr_FormatV()*
- *PyErr_GetExcInfo()*
- *PyErr_GetHandledException()*
- *PyErr_GetRaisedException()*
- *PyErr_GivenExceptionMatches()*
- *PyErr_NewException()*
- *PyErr_NewExceptionWithDoc()*
- *PyErr_NoMemory()*
- *PyErr_NormalizeException()*
- *PyErr_Occurred()*
- *PyErr_Print()*
- *PyErr_PrintEx()*
- PyErr_ProgramText()
- *PyErr_ResourceWarning()*
- *PyErr_Restore()*
- *PyErr_SetExcFromWindowsErr()*
- *PyErr_SetExcFromWindowsErrWithFilename()*
- *PyErr_SetExcFromWindowsErrWithFilenameObject()*
- *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
- *PyErr_SetExcInfo()*
- *PyErr_SetFromErrno()*
- *PyErr_SetFromErrnoWithFilename()*
- *PyErr_SetFromErrnoWithFilenameObject()*
- *PyErr_SetFromErrnoWithFilenameObjects()*
- *PyErr_SetFromWindowsErr()*
- *PyErr_SetFromWindowsErrWithFilename()*
- *PyErr_SetHandledException()*
- *PyErr_SetImportError()*

- *PyErr_SetImportErrorSubclass()*
- *PyErr_SetInterrupt()*
- *PyErr_SetInterruptEx()*
- *PyErr_SetNone()*
- *PyErr_SetObject()*
- *PyErr_SetRaisedException()*
- *PyErr_SetString()*
- *PyErr_SyntaxLocation()*
- *PyErr_SyntaxLocationEx()*
- *PyErr_WarnEx()*
- *PyErr_WarnExplicit()*
- *PyErr_WarnFormat()*
- *PyErr_WriteUnraisable()*
- *PyEval_AcquireThread()*
- *PyEval_EvalCode()*
- *PyEval_EvalCodeEx()*
- *PyEval_EvalFrame()*
- *PyEval_EvalFrameEx()*
- *PyEval_GetBuiltins()*
- *PyEval_GetFrame()*
- *PyEval_GetFrameBuiltins()*
- *PyEval_GetFrameGlobals()*
- *PyEval_GetFrameLocals()*
- *PyEval_GetFuncDesc()*
- *PyEval_GetFuncName()*
- *PyEval_GetGlobals()*
- *PyEval_GetLocals()*
- *PyEval_InitThreads()*
- *PyEval_ReleaseThread()*
- *PyEval_RestoreThread()*
- *PyEval_SaveThread()*
- *PyExc_ArithmeticError*
- *PyExc_AssertionError*
- *PyExc_AttributeError*
- *PyExc_BaseException*
- *PyExc_BaseExceptionGroup*
- *PyExc_BlockingIOError*
- *PyExc_BrokenPipeError*
- *PyExc_BufferError*

- *PyExc_BytesWarning*
- *PyExc_ChildProcessError*
- *PyExc_ConnectionAbortedError*
- *PyExc_ConnectionError*
- *PyExc_ConnectionRefusedError*
- *PyExc_ConnectionResetError*
- *PyExc_DeprecationWarning*
- *PyExc_EOFError*
- *PyExc_EncodingWarning*
- *PyExc_EnvironmentError*
- *PyExc_Exception*
- *PyExc_FileExistsError*
- *PyExc_FileNotFoundError*
- *PyExc_FloatingPointError*
- *PyExc_FutureWarning*
- *PyExc_GeneratorExit*
- *PyExc_IOError*
- *PyExc_ImportError*
- *PyExc_ImportWarning*
- *PyExc_IndentationError*
- *PyExc_IndexError*
- *PyExc_InterruptedError*
- *PyExc_IsADirectoryError*
- *PyExc_KeyError*
- *PyExc_KeyboardInterrupt*
- *PyExc_LookupError*
- *PyExc_MemoryError*
- *PyExc_ModuleNotFoundError*
- *PyExc_NameError*
- *PyExc_NotADirectoryError*
- *PyExc_NotImplementedError*
- *PyExc_OSError*
- *PyExc_OverflowError*
- *PyExc_PendingDeprecationWarning*
- *PyExc_PermissionError*
- *PyExc_ProcessLookupError*
- *PyExc_RecursionError*
- *PyExc_ReferenceError*
- *PyExc_ResourceWarning*

- *PyExc_RuntimeError*
- *PyExc_RuntimeWarning*
- *PyExc_StopAsyncIteration*
- *PyExc_StopIteration*
- *PyExc_SyntaxError*
- *PyExc_SyntaxWarning*
- *PyExc_SystemError*
- *PyExc_SystemExit*
- *PyExc_TabError*
- *PyExc_TimeoutError*
- *PyExc_TypeError*
- *PyExc_UnboundLocalError*
- *PyExc_UnicodeDecodeError*
- *PyExc_UnicodeEncodeError*
- *PyExc_UnicodeError*
- *PyExc_UnicodeTranslateError*
- *PyExc_UnicodeWarning*
- *PyExc_UserWarning*
- *PyExc_ValueError*
- *PyExc_Warning*
- *PyExc_WindowsError*
- *PyExc_ZeroDivisionError*
- *PyExceptionClass_Name()*
- *PyException_GetArgs()*
- *PyException_GetCause()*
- *PyException_GetContext()*
- *PyException_GetTraceback()*
- *PyException_SetArgs()*
- *PyException_SetCause()*
- *PyException_SetContext()*
- *PyException_SetTraceback()*
- *PyFile_FromFd()*
- *PyFile_GetLine()*
- *PyFile_WriteObject()*
- *PyFile_WriteString()*
- PyFilter_Type
- *PyFloat_AsDouble()*
- *PyFloat_FromDouble()*
- *PyFloat_FromString()*

- *PyFloat_GetInfo()*
- *PyFloat_GetMax()*
- *PyFloat_GetMin()*
- *PyFloat_Type*
- *PyFrameObject*
- *PyFrame_GetCode()*
- *PyFrame_GetLineNumber()*
- *PyFrozenSet_New()*
- *PyFrozenSet_Type*
- *PyGC_Collect()*
- *PyGC_Disable()*
- *PyGC_Enable()*
- *PyGC_IsEnabled()*
- *PyGILState_Ensure()*
- *PyGILState_GetThisThreadState()*
- *PyGILState_Release()*
- PyGILState_STATE
- *PyGetSetDef*
- PyGetSetDescr_Type
- *PyImport_AddModule()*
- *PyImport_AddModuleObject()*
- *PyImport_AddModuleRef()*
- *PyImport_AppendInittab()*
- *PyImport_ExecCodeModule()*
- *PyImport_ExecCodeModuleEx()*
- *PyImport_ExecCodeModuleObject()*
- *PyImport_ExecCodeModuleWithPathnames()*
- *PyImport_GetImporter()*
- *PyImport_GetMagicNumber()*
- *PyImport_GetMagicTag()*
- *PyImport_GetModule()*
- *PyImport_GetModuleDict()*
- *PyImport_Import()*
- *PyImport_ImportFrozenModule()*
- *PyImport_ImportFrozenModuleObject()*
- *PyImport_ImportModule()*
- *PyImport_ImportModuleLevel()*
- *PyImport_ImportModuleLevelObject()*
- *PyImport_ImportModuleNoBlock()*

- *PyImport_ReloadModule()*
- *PyIndex_Check()*
- *PyInterpreterState*
- *PyInterpreterState_Clear()*
- *PyInterpreterState_Delete()*
- *PyInterpreterState_Get()*
- *PyInterpreterState_GetDict()*
- *PyInterpreterState_GetID()*
- *PyInterpreterState_New()*
- *PyIter_Check()*
- *PyIter_Next()*
- *PyIter_NextItem()*
- *PyIter_Send()*
- PyListIter_Type
- PyListRevIter_Type
- *PyList_Append()*
- *PyList_AsTuple()*
- *PyList_GetItem()*
- *PyList_GetItemRef()*
- *PyList_GetSlice()*
- *PyList_Insert()*
- *PyList_New()*
- *PyList_Reverse()*
- *PyList_SetItem()*
- *PyList_SetSlice()*
- *PyList_Size()*
- *PyList_Sort()*
- *PyList_Type*
- *PyLongObject*
- PyLongRangeIter_Type
- *PyLong_AsDouble()*
- *PyLong_AsInt()*
- *PyLong_AsInt32()*
- *PyLong_AsInt64()*
- *PyLong_AsLong()*
- *PyLong_AsLongAndOverflow()*
- *PyLong_AsLongLong()*
- *PyLong_AsLongLongAndOverflow()*
- *PyLong_AsNativeBytes()*

- *PyLong_AsSize_t()*
- *PyLong_AsSsize_t()*
- *PyLong_AsUInt32()*
- *PyLong_AsUInt64()*
- *PyLong_AsUnsignedLong()*
- *PyLong_AsUnsignedLongLong()*
- *PyLong_AsUnsignedLongLongMask()*
- *PyLong_AsUnsignedLongMask()*
- *PyLong_AsVoidPtr()*
- *PyLong_FromDouble()*
- *PyLong_FromInt32()*
- *PyLong_FromInt64()*
- *PyLong_FromLong()*
- *PyLong_FromLongLong()*
- *PyLong_FromNativeBytes()*
- *PyLong_FromSize_t()*
- *PyLong_FromSsize_t()*
- *PyLong_FromString()*
- *PyLong_FromUInt32()*
- *PyLong_FromUInt64()*
- *PyLong_FromUnsignedLong()*
- *PyLong_FromUnsignedLongLong()*
- *PyLong_FromUnsignedNativeBytes()*
- *PyLong_FromVoidPtr()*
- *PyLong_GetInfo()*
- *PyLong_Type*
- PyMap_Type
- *PyMapping_Check()*
- *PyMapping_GetItemString()*
- *PyMapping_GetOptionalItem()*
- *PyMapping_GetOptionalItemString()*
- *PyMapping_HasKey()*
- *PyMapping_HasKeyString()*
- *PyMapping_HasKeyStringWithError()*
- *PyMapping_HasKeyWithError()*
- *PyMapping_Items()*
- *PyMapping_Keys()*
- *PyMapping_Length()*
- *PyMapping_SetItemString()*

- *PyMapping_Size()*
- *PyMapping_Values()*
- *PyMem_Calloc()*
- *PyMem_Free()*
- *PyMem_Malloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*
- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_Realloc()*
- *PyMemberDef*
- PyMemberDescr_Type
- *PyMember_GetOne()*
- *PyMember_SetOne()*
- *PyMemoryView_FromBuffer()*
- *PyMemoryView_FromMemory()*
- *PyMemoryView_FromObject()*
- *PyMemoryView_GetContiguous()*
- PyMemoryView_Type
- *PyMethodDef*
- PyMethodDescr_Type
- *PyModuleDef*
- PyModuleDef_Base
- *PyModuleDef_Init()*
- PyModuleDef_Type
- *PyModule_Add()*
- *PyModule_AddFunctions()*
- *PyModule_AddIntConstant()*
- *PyModule_AddObject()*
- *PyModule_AddObjectRef()*
- *PyModule_AddStringConstant()*
- *PyModule_AddType()*
- *PyModule_Create2()*
- *PyModule_ExecDef()*
- *PyModule_FromDefAndSpec2()*
- *PyModule_GetDef()*
- *PyModule_GetDict()*
- *PyModule_GetFilename()*
- *PyModule_GetFilenameObject()*

- *PyModule_GetName()*
- *PyModule_GetNameObject()*
- *PyModule_GetState()*
- *PyModule_New()*
- *PyModule_NewObject()*
- *PyModule_SetDocString()*
- *PyModule_Type*
- *PyNumber_Absolute()*
- *PyNumber_Add()*
- *PyNumber_And()*
- *PyNumber_AsSsize_t()*
- *PyNumber_Check()*
- *PyNumber_Divmod()*
- *PyNumber_Float()*
- *PyNumber_FloorDivide()*
- *PyNumber_InPlaceAdd()*
- *PyNumber_InPlaceAnd()*
- *PyNumber_InPlaceFloorDivide()*
- *PyNumber_InPlaceLshift()*
- *PyNumber_InPlaceMatrixMultiply()*
- *PyNumber_InPlaceMultiply()*
- *PyNumber_InPlaceOr()*
- *PyNumber_InPlacePower()*
- *PyNumber_InPlaceRemainder()*
- *PyNumber_InPlaceRshift()*
- *PyNumber_InPlaceSubtract()*
- *PyNumber_InPlaceTrueDivide()*
- *PyNumber_InPlaceXor()*
- *PyNumber_Index()*
- *PyNumber_Invert()*
- *PyNumber_Long()*
- *PyNumber_Lshift()*
- *PyNumber_MatrixMultiply()*
- *PyNumber_Multiply()*
- *PyNumber_Negative()*
- *PyNumber_Or()*
- *PyNumber_Positive()*
- *PyNumber_Power()*
- *PyNumber_Remainder()*

- *PyNumber_Rshift()*
- *PyNumber_Subtract()*
- *PyNumber_ToBase()*
- *PyNumber_TrueDivide()*
- *PyNumber_Xor()*
- *PyOS_AfterFork()*
- *PyOS_AfterFork_Child()*
- *PyOS_AfterFork_Parent()*
- *PyOS_BeforeFork()*
- *PyOS_CheckStack()*
- *PyOS_FSPath()*
- *PyOS_InputHook*
- PyOS_InterruptOccurred()
- *PyOS_double_to_string()*
- *PyOS_getsig()*
- PyOS_mystricmp()
- PyOS_mystrnicmp()
- *PyOS_setsig()*
- *PyOS_sighandler_t*
- *PyOS_snprintf()*
- *PyOS_string_to_double()*
- *PyOS_strtol()*
- *PyOS_strtoul()*
- *PyOS_vsnprintf()*
- *PyObject*
- *PyObject.ob_refcnt*
- *PyObject.ob_type*
- *PyObject_ASCII()*
- *PyObject_AsFileDescriptor()*
- *PyObject_Bytes()*
- *PyObject_Call()*
- *PyObject_CallFunction()*
- *PyObject_CallFunctionObjArgs()*
- *PyObject_CallMethod()*
- *PyObject_CallMethodObjArgs()*
- *PyObject_CallNoArgs()*
- *PyObject_CallObject()*
- *PyObject_Calloc()*
- *PyObject_CheckBuffer()*

- *PyObject_ClearWeakRefs()*
- *PyObject_CopyData()*
- *PyObject_DelAttr()*
- *PyObject_DelAttrString()*
- *PyObject_DelItem()*
- *PyObject_DelItemString()*
- *PyObject_Dir()*
- *PyObject_Format()*
- *PyObject_Free()*
- *PyObject_GC_Del()*
- *PyObject_GC_IsFinalized()*
- *PyObject_GC_IsTracked()*
- *PyObject_GC_Track()*
- *PyObject_GC_UnTrack()*
- *PyObject_GenericGetAttr()*
- *PyObject_GenericGetDict()*
- *PyObject_GenericSetAttr()*
- *PyObject_GenericSetDict()*
- *PyObject_GetAIter()*
- *PyObject_GetAttr()*
- *PyObject_GetAttrString()*
- *PyObject_GetBuffer()*
- *PyObject_GetItem()*
- *PyObject_GetIter()*
- *PyObject_GetOptionalAttr()*
- *PyObject_GetOptionalAttrString()*
- *PyObject_GetTypeData()*
- *PyObject_HasAttr()*
- *PyObject_HasAttrString()*
- *PyObject_HasAttrStringWithError()*
- *PyObject_HasAttrWithError()*
- *PyObject_Hash()*
- *PyObject_HashNotImplemented()*
- *PyObject_Init()*
- *PyObject_InitVar()*
- *PyObject_IsInstance()*
- *PyObject_IsSubclass()*
- *PyObject_IsTrue()*
- *PyObject_Length()*

- *PyObject_Malloc()*
- *PyObject_Not()*
- *PyObject_Realloc()*
- *PyObject_Repr()*
- *PyObject_RichCompare()*
- *PyObject_RichCompareBool()*
- *PyObject_SelfIter()*
- *PyObject_SetAttr()*
- *PyObject_SetAttrString()*
- *PyObject_SetItem()*
- *PyObject_Size()*
- *PyObject_Str()*
- *PyObject_Type()*
- *PyObject_Vectorcall()*
- *PyObject_VectorcallMethod()*
- *PyProperty_Type*
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type
- *PySeqIter_New()*
- *PySeqIter_Type*
- *PySequence_Check()*
- *PySequence_Concat()*
- *PySequence_Contains()*
- *PySequence_Count()*
- *PySequence_DelItem()*
- *PySequence_DelSlice()*
- *PySequence_Fast()*
- *PySequence_GetItem()*
- *PySequence_GetSlice()*
- *PySequence_In()*
- *PySequence_InPlaceConcat()*
- *PySequence_InPlaceRepeat()*
- *PySequence_Index()*
- *PySequence_Length()*
- *PySequence_List()*
- *PySequence_Repeat()*
- *PySequence_SetItem()*
- *PySequence_SetSlice()*

- *PySequence_Size()*
- *PySequence_Tuple()*
- PySetIter_Type
- *PySet_Add()*
- *PySet_Clear()*
- *PySet_Contains()*
- *PySet_Discard()*
- *PySet_New()*
- *PySet_Pop()*
- *PySet_Size()*
- *PySet_Type*
- *PySlice_AdjustIndices()*
- *PySlice_GetIndices()*
- *PySlice_GetIndicesEx()*
- *PySlice_New()*
- *PySlice_Type*
- *PySlice_Unpack()*
- *PyState_AddModule()*
- *PyState_FindModule()*
- *PyState_RemoveModule()*
- *PyStructSequence_Desc*
- *PyStructSequence_Field*
- *PyStructSequence_GetItem()*
- *PyStructSequence_New()*
- *PyStructSequence_NewType()*
- *PyStructSequence_SetItem()*
- *PyStructSequence_UnnamedField*
- PySuper_Type
- *PySys_Audit()*
- *PySys_AuditTuple()*
- *PySys_FormatStderr()*
- *PySys_FormatStdout()*
- *PySys_GetObject()*
- *PySys_GetXOptions()*
- *PySys_ResetWarnOptions()*
- *PySys_SetArgv()*
- *PySys_SetArgvEx()*
- *PySys_SetObject()*
- *PySys_WriteStderr()*

- *PySys_WriteStdout()*
- *PyThreadState*
- *PyThreadState_Clear()*
- *PyThreadState_Delete()*
- *PyThreadState_Get()*
- *PyThreadState_GetDict()*
- *PyThreadState_GetFrame()*
- *PyThreadState_GetID()*
- *PyThreadState_GetInterpreter()*
- *PyThreadState_New()*
- *PyThreadState_SetAsyncExc()*
- *PyThreadState_Swap()*
- PyThread_GetInfo()
- *PyThread_ReInitTLS()*
- PyThread_acquire_lock()
- PyThread_acquire_lock_timed()
- PyThread_allocate_lock()
- *PyThread_create_key()*
- *PyThread_delete_key()*
- *PyThread_delete_key_value()*
- PyThread_exit_thread()
- PyThread_free_lock()
- *PyThread_get_key_value()*
- PyThread_get_stacksize()
- PyThread_get_thread_ident()
- PyThread_get_thread_native_id()
- PyThread_init_thread()
- PyThread_release_lock()
- *PyThread_set_key_value()*
- PyThread_set_stacksize()
- PyThread_start_new_thread()
- *PyThread_tss_alloc()*
- *PyThread_tss_create()*
- *PyThread_tss_delete()*
- *PyThread_tss_free()*
- *PyThread_tss_get()*
- *PyThread_tss_is_created()*
- *PyThread_tss_set()*
- PyTraceBack_Here()

- PyTraceBack_Print()
- PyTraceBack_Type
- PyTupleIter_Type
- *PyTuple_GetItem()*
- *PyTuple_GetSlice()*
- *PyTuple_New()*
- *PyTuple_Pack()*
- *PyTuple_SetItem()*
- *PyTuple_Size()*
- *PyTuple_Type*
- *PyTypeObject*
- *PyType_ClearCache()*
- *PyType_Freeze()*
- *PyType_FromMetaclass()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetBaseByToken()*
- *PyType_GetFlags()*
- *PyType_GetFullyQualifiedName()*
- *PyType_GetModule()*
- *PyType_GetModuleByDef()*
- *PyType_GetModuleName()*
- *PyType_GetModuleState()*
- *PyType_GetName()*
- *PyType_GetQualName()*
- *PyType_GetSlot()*
- *PyType_GetTypeDataSize()*
- *PyType_IsSubtype()*
- *PyType_Modified()*
- *PyType_Ready()*
- *PyType_Slot*
- *PyType_Spec*
- *PyType_Type*
- *PyUnicodeDecodeError_Create()*
- *PyUnicodeDecodeError_GetEncoding()*
- *PyUnicodeDecodeError_GetEnd()*

- *PyUnicodeDecodeError_GetObject()*
- *PyUnicodeDecodeError_GetReason()*
- *PyUnicodeDecodeError_GetStart()*
- *PyUnicodeDecodeError_SetEnd()*
- *PyUnicodeDecodeError_SetReason()*
- *PyUnicodeDecodeError_SetStart()*
- *PyUnicodeEncodeError_GetEncoding()*
- *PyUnicodeEncodeError_GetEnd()*
- *PyUnicodeEncodeError_GetObject()*
- *PyUnicodeEncodeError_GetReason()*
- *PyUnicodeEncodeError_GetStart()*
- *PyUnicodeEncodeError_SetEnd()*
- *PyUnicodeEncodeError_SetReason()*
- *PyUnicodeEncodeError_SetStart()*
- *PyUnicodeIter_Type*
- *PyUnicodeTranslateError_GetEnd()*
- *PyUnicodeTranslateError_GetObject()*
- *PyUnicodeTranslateError_GetReason()*
- *PyUnicodeTranslateError_GetStart()*
- *PyUnicodeTranslateError_SetEnd()*
- *PyUnicodeTranslateError_SetReason()*
- *PyUnicodeTranslateError_SetStart()*
- *PyUnicode_Append()*
- *PyUnicode_AppendAndDel()*
- *PyUnicode_AsASCIIString()*
- *PyUnicode_AsCharmapString()*
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()
- *PyUnicode_AsEncodedString()*
- PyUnicode_AsEncodedUnicode()
- *PyUnicode_AsLatin1String()*
- *PyUnicode_AsMBCSString()*
- *PyUnicode_AsRawUnicodeEscapeString()*
- *PyUnicode_AsUCS4()*
- *PyUnicode_AsUCS4Copy()*
- *PyUnicode_AsUTF16String()*
- *PyUnicode_AsUTF32String()*
- *PyUnicode_AsUTF8AndSize()*

- *PyUnicode_AsUTF8String()*
- *PyUnicode_AsUnicodeEscapeString()*
- *PyUnicode_AsWideChar()*
- *PyUnicode_AsWideCharString()*
- *PyUnicode_BuildEncodingMap()*
- *PyUnicode_Compare()*
- *PyUnicode_CompareWithASCIIString()*
- *PyUnicode_Concat()*
- *PyUnicode_Contains()*
- *PyUnicode_Count()*
- *PyUnicode_Decode()*
- *PyUnicode_DecodeASCII()*
- *PyUnicode_DecodeCharmap()*
- *PyUnicode_DecodeCodePageStateful()*
- *PyUnicode_DecodeFSDefault()*
- *PyUnicode_DecodeFSDefaultAndSize()*
- *PyUnicode_DecodeLatin1()*
- *PyUnicode_DecodeLocale()*
- *PyUnicode_DecodeLocaleAndSize()*
- *PyUnicode_DecodeMBCS()*
- *PyUnicode_DecodeMBCSStateful()*
- *PyUnicode_DecodeRawUnicodeEscape()*
- *PyUnicode_DecodeUTF16()*
- *PyUnicode_DecodeUTF16Stateful()*
- *PyUnicode_DecodeUTF32()*
- *PyUnicode_DecodeUTF32Stateful()*
- *PyUnicode_DecodeUTF7()*
- *PyUnicode_DecodeUTF7Stateful()*
- *PyUnicode_DecodeUTF8()*
- *PyUnicode_DecodeUTF8Stateful()*
- *PyUnicode_DecodeUnicodeEscape()*
- *PyUnicode_EncodeCodePage()*
- *PyUnicode_EncodeFSDefault()*
- *PyUnicode_EncodeLocale()*
- *PyUnicode_Equal()*
- *PyUnicode_EqualToUTF8()*
- *PyUnicode_EqualToUTF8AndSize()*
- *PyUnicode_FSConverter()*
- *PyUnicode_FSDecoder()*

- *PyUnicode_Find()*
- *PyUnicode_FindChar()*
- *PyUnicode_Format()*
- *PyUnicode_FromEncodedObject()*
- *PyUnicode_FromFormat()*
- *PyUnicode_FromFormatV()*
- *PyUnicode_FromObject()*
- *PyUnicode_FromOrdinal()*
- *PyUnicode_FromString()*
- *PyUnicode_FromStringAndSize()*
- *PyUnicode_FromWideChar()*
- *PyUnicode_GetDefaultEncoding()*
- *PyUnicode_GetLength()*
- *PyUnicode_InternFromString()*
- *PyUnicode_InternInPlace()*
- *PyUnicode_IsIdentifier()*
- *PyUnicode_Join()*
- *PyUnicode_Partition()*
- *PyUnicode_RPartition()*
- *PyUnicode_RSplit()*
- *PyUnicode_ReadChar()*
- *PyUnicode_Replace()*
- *PyUnicode_Resize()*
- *PyUnicode_RichCompare()*
- *PyUnicode_Split()*
- *PyUnicode_Splitlines()*
- *PyUnicode_Substring()*
- *PyUnicode_Tailmatch()*
- *PyUnicode_Translate()*
- *PyUnicode_Type*
- *PyUnicode_WriteChar()*
- *PyVarObject*
- PyVarObject.ob_base
- *PyVarObject.ob_size*
- *PyVectorcall_Call()*
- *PyVectorcall_NARGS()*
- PyWeakReference
- *PyWeakref_GetObject()*
- *PyWeakref_GetRef()*

- *PyWeakref_NewProxy()*
- *PyWeakref_NewRef()*
- PyWrapperDescr_Type
- *PyWrapper_New()*
- PyZip_Type
- *Py_AddPendingCall()*
- *Py_AtExit()*
- *Py_BEGIN_ALLOW_THREADS*
- *Py_BLOCK_THREADS*
- *Py_BuildValue()*
- *Py_BytesMain()*
- *Py_CompileString()*
- *Py_DecRef()*
- *Py_DecodeLocale()*
- *Py_END_ALLOW_THREADS*
- *Py_EncodeLocale()*
- *Py_EndInterpreter()*
- *Py_EnterRecursiveCall()*
- *Py_Exit()*
- *Py_FatalError()*
- Py_FileSystemDefaultEncodeErrors
- Py_FileSystemDefaultEncoding
- *Py_Finalize()*
- *Py_FinalizeEx()*
- *Py_GenericAlias()*
- *Py_GenericAliasType*
- *Py_GetBuildInfo()*
- *Py_GetCompiler()*
- *Py_GetConstant()*
- *Py_GetConstantBorrowed()*
- *Py_GetCopyright()*
- *Py_GetExecPrefix()*
- *Py_GetPath()*
- *Py_GetPlatform()*
- *Py_GetPrefix()*
- *Py_GetProgramFullPath()*
- *Py_GetProgramName()*
- *Py_GetPythonHome()*
- Py_GetRecursionLimit()

- *Py_GetVersion()*
- Py_HasFileSystemDefaultEncoding
- *Py_IncRef()*
- *Py_Initialize()*
- *Py_InitializeEx()*
- *Py_Is()*
- *Py_IsFalse()*
- *Py_IsFinalizing()*
- *Py_IsInitialized()*
- *Py_IsNone()*
- *Py_IsTrue()*
- *Py_LeaveRecursiveCall()*
- *Py_Main()*
- Py_MakePendingCalls()
- *Py_NewInterpreter()*
- *Py_NewRef()*
- *Py_PACK_FULL_VERSION()*
- *Py_PACK_VERSION()*
- *Py_REFCNT()*
- *Py_ReprEnter()*
- *Py_ReprLeave()*
- *Py_SetProgramName()*
- *Py_SetPythonHome()*
- Py_SetRecursionLimit()
- *Py_TYPE()*
- *Py_UCS4*
- *Py_UNBLOCK_THREADS*
- Py_UTF8Mode
- *Py_VaBuildValue()*
- *Py_Version*
- *Py_XNewRef()*
- *Py_buffer*
- Py_intptr_t
- *Py_ssize_t*
- Py_uintptr_t
- *allocfunc*
- *binaryfunc*
- *descrgetfunc*
- *descrsetfunc*

- *destructor*
- *getattrfunc*
- *getattrofunc*
- *getbufferproc*
- *getiterfunc*
- *getter*
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*
- *objobjproc*
- *releasebufferproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*
- *setter*
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*
- *traverseproc*
- *unaryfunc*
- *vectorcallfunc*
- *visitproc*

CHAPTER 3

매우 고수준 계층

이 장의 함수들은 파일이나 버퍼에 제공된 파이썬 소스 코드를 실행할 수 있도록 하지만, 인터프리터와 더 세밀한 방식으로 상호 작용하도록 하지는 않습니다.

이러한 함수 중 일부는 문법의 시작 기호를 매개 변수로 받아들입니다. 사용 가능한 시작 기호는 *Py_eval_input*, *Py_file_input* 및 *Py_single_input* 입니다. 이것들은 이들을 매개 변수로 받아들이는 함수 뒤에 설명됩니다.

또한 이 함수 중 일부는 `FILE*` 매개 변수를 취합니다. 주의해서 다루어야 할 한 가지 문제는 다른 C 라이브러리의 `FILE` 구조체가 다르고, 호환되지 않을 수 있다는 것입니다. (적어도) 윈도우에서는, 동적으로 링크된 확장에서 실제로 다른 라이브러리를 사용할 수 있어서, `FILE*` 매개 변수가 파이썬 런타임이 사용하고 있는 것과 같은 라이브러리에서 만들어진 것이 확실할 때만 이러한 함수에 전달되도록 주의해야 합니다.

int **PyRun_AnyFile**(FILE *fp, const char *filename)

아래 *PyRun_AnyFileExFlags()* 의 단순화된 인터페이스입니다. *closeit*은 `0`으로 *flags*는 `NULL`로 설정된 상태로 남겨둡니다.

int **PyRun_AnyFileFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

아래 *PyRun_AnyFileExFlags()* 의 단순화된 인터페이스입니다. 이것은 *closeit* 인자를 `0`으로 설정된 상태로 남겨둡니다.

int **PyRun_AnyFileEx**(FILE *fp, const char *filename, int closeit)

아래 *PyRun_AnyFileExFlags()* 의 단순화된 인터페이스입니다. 이것은 *flags* 인자를 `NULL`로 설정된 상태로 남겨둡니다.

int **PyRun_AnyFileExFlags**(FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

*fp*가 대화식 장치(콘솔이나 터미널 입력이나 유닉스 의사 터미널)와 연결된 파일을 가리키면, *PyRun_InteractiveLoop()*의 값을 반환하고, 그렇지 않으면 *PyRun_SimpleFile()*의 결과를 반환합니다. *filename*은 파일 시스템 인코딩(`sys.getfilesystemencoding()`)으로 디코딩됩니다. *filename*이 `NULL`이면, 이 함수는 파일명으로 `"???"`를 사용합니다. *closeit*이 참이면, `PyRun_SimpleFileExFlags()`이 반환하기 전에 파일이 닫힙니다.

int **PyRun_SimpleString**(const char *command)

아래 *PyRun_SimpleStringFlags()* 의 단순화된 인터페이스입니다. *PyCompilerFlags** 인자를 `NULL`로 설정된 상태로 남겨둡니다.

int **PyRun_SimpleStringFlags**(const char *command, *PyCompilerFlags* *flags)

flags 인자에 따라 `__main__` 모듈에서 *command*에 있는 파이썬 소스 코드를 실행합니다. `__main__`이 존재하지 않으면 만듭니다. 성공하면 `0`을, 예외가 발생하면 `-1`을 반환합니다. 에러가 있으면, 예외 정보를 얻을 방법이 없습니다. *flags*의 의미는 아래를 참조하십시오.

처리되지 않은 SystemExit가 발생하면, 이 함수는 *PyConfig.inspect*가 0인 한 -1을 반환하지 않고 프로세스를 종료함에 유의하십시오.

int **PyRun_SimpleFile**(FILE *fp, const char *filename)

아래 *PyRun_SimpleFileExFlags()* 의 단순화된 인터페이스입니다. *closeit*을 0으로, *flags*를 NULL로 설정된 상태로 남겨둡니다.

int **PyRun_SimpleFileEx**(FILE *fp, const char *filename, int closeit)

아래 *PyRun_SimpleFileExFlags()* 의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 남겨둡니다.

int **PyRun_SimpleFileExFlags**(FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

PyRun_SimpleStringFlags() 와 비슷하지만, 메모리에 있는 문자열 대신 *fp*에서 파이썬 소스 코드를 읽습니다. *filename*은 파일의 이름이어야 하며, 파일시스템 인코딩과 에러 처리기로 디코딩됩니다. *closeit*이 참이면 PyRun_SimpleFileExFlags() 가 반환하기 전에 파일이 닫힙니다.

> **참고**
>
> 윈도우에서, *fp*는 바이너리 모드로 열어야 합니다 (예를 들어 fopen(filename, "rb")). 그렇지 않으면, 파이썬은 LF 줄 종료가 있는 스크립트 파일을 올바르게 처리하지 못할 수 있습니다.

int **PyRun_InteractiveOne**(FILE *fp, const char *filename)

아래 *PyRun_InteractiveOneFlags()* 의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 남겨둡니다.

int **PyRun_InteractiveOneFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

flags 인자에 따라 대화식 장치와 연관된 파일에서 단일 문장을 읽고 실행합니다. sys.ps1과 sys.ps2를 사용하여 사용자에게 프롬프트 합니다. *filename*은 파일시스템 인코딩과 에러 처리기로 디코딩됩니다.

입력이 성공적으로 실행될 때 0을, 예외가 있으면 -1을, 또는 구문 분석 에러가 있으면 파이썬의 일부로 배포된 errcode.h 인클루드 파일에 있는 에러 코드를 반환합니다. (errcode.h는 Python.h에서 인클루드하지 않기 때문에 필요하면 특별히 인클루드해야 함에 유의하십시오.)

int **PyRun_InteractiveLoop**(FILE *fp, const char *filename)

아래 *PyRun_InteractiveLoopFlags()* 의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 남겨둡니다.

int **PyRun_InteractiveLoopFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

EOF에 도달할 때까지 대화식 장치와 연관된 파일에서 문장을 읽고 실행합니다. sys.ps1과 sys.ps2를 사용하여 사용자에게 프롬프트 합니다. *filename*은 파일시스템 인코딩과 에러 처리기로 디코딩됩니다. EOF에서 0을 반환하거나, 실패하면 음수를 반환합니다.

int (***PyOS_InputHook**)(void)

Part of the 안정 ABI. 프로토타입 int func(void) 인 함수를 가리키도록 설정할 수 있습니다. 이 함수는 파이썬의 인터프리터 프롬프트가 유휴 상태가 되고 터미널에서 사용자 입력을 기다리려고 할 때 호출됩니다. 반환 값은 무시됩니다. 이 훅을 재정의하는 것은 파이썬 소스 코드의 Modules/_tkinter.c에서 한 것처럼 인터프리터의 프롬프트를 다른 이벤트 루프와 통합하는 데 사용될 수 있습니다.

버전 3.12에서 변경: This function is only called from the *main interpreter*.

char *(***PyOS_ReadlineFunctionPointer**)(FILE*, FILE*, const char*)

프로토타입 char *func(FILE *stdin, FILE *stdout, char *prompt) 인 함수를 가리키도록 설정하여, 인터프리터의 프롬프트에서 단일 입력 줄을 읽는 데 사용되는 기본 함수를 재정의할 수 있습니다. 이 함수는 NULL이 아니면 문자열 *prompt*를 출력한 다음 제공된 표준 입력 파일에서 입력 줄을 읽고 결과 문자열을 반환할 것이라고 기대됩니다. 예를 들어, readline 모듈은 이 훅을 설정하여 줄 편집과 탭 완성 기능을 제공합니다.

결과는 *PyMem_RawMalloc()* 이나 *PyMem_RawRealloc()* 으로 할당된 문자열 이거나, 에러가 발생했으면 NULL이어야 합니다.

버전 3.4에서 변경: 결과는 *PyMem_Malloc()*이나 *PyMem_Realloc()*으로 할당하는 대신, *PyMem_RawMalloc()*이나 *PyMem_RawRealloc()*으로 할당해야 합니다.

버전 3.12에서 변경: This function is only called from the *main interpreter*.

PyObject ***PyRun_String**(const char *str, int start, *PyObject* *globals, *PyObject* *locals)

반환값: 새 참조. 아래 *PyRun_StringFlags()*의 단순화된 인터페이스입니다. 이것은 *flags*를 NULL로 설정된 상태로 남겨둡니다.

PyObject ***PyRun_StringFlags**(const char *str, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

반환값: 새 참조. *flags*로 지정된 컴파일러 플래그를 사용하여 *globals*과 *locals* 객체로 지정된 컨텍스트에서 *str*에서 파이썬 소스 코드를 실행합니다. *globals*는 딕셔너리이어야 합니다. *locals*는 매핑 프로토콜을 구현하는 모든 객체가 될 수 있습니다. 매개 변수 *start*는 소스 코드를 구문 분석하는 데 사용해야 하는 시작 토큰을 지정합니다.

코드를 실행한 결과를 파이썬 객체로 반환하거나, 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyRun_File**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals)

반환값: 새 참조. 아래 *PyRun_FileExFlags()*의 단순화된 인터페이스입니다. *closeit*을 0으로, *flags*를 NULL로 설정된 상태로 남겨둡니다.

PyObject ***PyRun_FileEx**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit)

반환값: 새 참조. 아래 *PyRun_FileExFlags()*의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 남겨둡니다.

PyObject ***PyRun_FileFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

반환값: 새 참조. 아래 *PyRun_FileExFlags()*의 단순화된 인터페이스입니다. *closeit*을 0으로 설정된 상태로 남겨둡니다.

PyObject ***PyRun_FileExFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit, *PyCompilerFlags* *flags)

반환값: 새 참조. *PyRun_StringFlags()*와 유사하지만, 파이썬 소스 코드는 메모리에 있는 문자열 대신 *fp*에서 읽습니다. *filename*은 파일명이어야 하며 파일시스템 인코딩과 에러 처리기로 디코딩됩니다. *closeit*이 참이면 *PyRun_FileExFlags()*가 반환되기 전에 파일이 닫힙니다.

PyObject ***Py_CompileString**(const char *str, const char *filename, int start)

반환값: 새 참조. *Part of the* 안정 ABI. 아래 *Py_CompileStringFlags()* 의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 유지합니다.

PyObject ***Py_CompileStringFlags**(const char *str, const char *filename, int start, *PyCompilerFlags* *flags)

반환값: 새 참조. 아래 *Py_CompileStringExFlags()* 의 단순화된 인터페이스입니다. *optimize*를 -1로 설정된 상태로 유지합니다.

PyObject ***Py_CompileStringObject**(const char *str, *PyObject* *filename, int start, *PyCompilerFlags* *flags, int optimize)

반환값: 새 참조. *str*에 있는 파이썬 소스 코드를 구문 분석하고 컴파일하여 결과 코드 객체를 반환합니다. 시작 토큰은 *start*로 주어집니다; 이것은 컴파일될 수 있는 코드를 제한하는 데 사용될 수 있으며 *Py_eval_input*, *Py_file_input* 또는 *Py_single_input*이어야 합니다. *filename*으로 지정된 파일명은 코드 객체를 구성하는 데 사용되며 트레이스백이나 SyntaxError 예외 메시지에 나타날 수 있습니다. 코드를 구문 분석할 수 없거나 컴파일할 수 없으면 NULL을 반환합니다.

정수 *optimize*는 컴파일러의 최적화 수준을 지정합니다. -1 값은 -O 옵션으로 주어진 것처럼 인터프리터의 최적화 수준을 선택합니다. 명시적 수준은 0 (최적화 없음; __debug__가 참), 1 (어서션이 제거되고 __debug__가 거짓) 또는 2 (독스트링도 제거됩니다) 입니다.

Added in version 3.4.

PyObject ***Py_CompileStringExFlags**(const char *str, const char *filename, int start, *PyCompilerFlags* *flags, int optimize)

반환값: 새 참조. *Py_CompileStringObject()*와 유사하지만, *filename*은 파일시스템 인코딩과 에러 처리기로 디코딩된 바이트 문자열입니다.

Added in version 3.2.

PyObject ***PyEval_EvalCode**(*PyObject* *co, *PyObject* *globals, *PyObject* *locals)

반환값: 새 참조. *Part of the* 안정 ABI. 이것은 코드 객체와 전역 변수 및 지역 변수만 있는, *PyEval_EvalCodeEx()*의 단순화된 인터페이스입니다. 다른 인자는 NULL로 설정됩니다.

PyObject ***PyEval_EvalCodeEx**(*PyObject* *co, *PyObject* *globals, *PyObject* *locals, *PyObject* *const *args, int argcount, *PyObject* *const *kws, int kwcount, *PyObject* *const *defs, int defcount, *PyObject* *kwdefs, *PyObject* *closure)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 평가를 위한 특정 환경에서, 미리 컴파일된 코드 객체를 평가합니다. 이 환경은 전역 변수의 딕셔너리, 지역 변수의 매핑 객체, 인자의 배열, 키워드와 기본값, 키워드 전용 인자의 기본값 딕셔너리 및 셀의 클로저 튜플로 구성됩니다.

PyObject ***PyEval_EvalFrame**(*PyFrameObject* *f)

반환값: 새 참조. *Part of the* 안정 ABI. 실행 프레임을 평가합니다. 이전 버전과의 호환성을 위한, *PyEval_EvalFrameEx()*의 단순화된 인터페이스입니다.

PyObject ***PyEval_EvalFrameEx**(*PyFrameObject* *f, int throwflag)

반환값: 새 참조. *Part of the* 안정 ABI. 이것은 파이썬 인터프리트의 메인, 꾸미지 않은 함수입니다. 실행 프레임 *f*와 연관된 코드 객체가 실행됩니다. 필요에 따라 바이트 코드를 해석하고 호출을 실행합니다. 추가 *throwflag* 매개 변수는 대체로 무시할 수 있습니다 - 참이면, 예외가 즉시 발생하도록 합니다; 제너레이터 객체의 `throw()` 메서드에 사용됩니다.

버전 3.4에서 변경: 이 함수는 이제 활성 예외를 조용히 버리지 않았는지 확인하도록 도우려고 디버그 어서션을 포함합니다.

int **PyEval_MergeCompilerFlags**(*PyCompilerFlags* *cf)

이 함수는 현재 평가 프레임의 플래그를 변경하고, 성공하면 참을, 실패하면 거짓을 반환합니다.

int **Py_eval_input**

격리된 표현식을 위한 파이썬 문법의 시작 기호; *Py_CompileString()*과 함께 사용합니다.

int **Py_file_input**

파일이나 다른 소스에서 읽은 문장의 시퀀스를 위한 파이썬 문법의 시작 기호; *Py_CompileString()*과 함께 사용합니다. 임의로 긴 파이썬 소스 코드를 컴파일할 때 사용하는 기호입니다.

int **Py_single_input**

단일 문장을 위한 파이썬 문법의 시작 기호; *Py_CompileString()*과 함께 사용합니다. 대화식 인터프리터 루프에 사용되는 기호입니다.

struct **PyCompilerFlags**

이것은 컴파일러 플래그를 담는 데 사용되는 구조체입니다. 코드가 컴파일되기만 하는 경우 `int flags`로 전달되고, 코드가 실행되는 경우 `PyCompilerFlags *flags`로 전달됩니다. 이 경우, `from __future__ import`는 *flags*를 수정할 수 있습니다.

`PyCompilerFlags *flags`가 NULL일 때마다, *cf_flags*는 0과 같다고 취급되며, `from __future__ import`로 인한 수정은 버립니다.

int **cf_flags**

컴파일러 플래그.

int **cf_feature_version**

*cf_feature_version*은 부 파이썬 버전입니다. `PY_MINOR_VERSION`으로 초기화되어야 합니다.

이 필드는 기본적으로 무시되며, `PyCF_ONLY_AST` 플래그가 *cf_flags*에 설정된 경우에만 사용됩니다.

버전 3.8에서 변경: *cf_feature_version* 필드를 추가했습니다.

The available compiler flags are accessible as macros:

PyCF_ALLOW_TOP_LEVEL_AWAIT

PyCF_ONLY_AST

PyCF_OPTIMIZED_AST

PyCF_TYPE_COMMENTS

See compiler flags in documentation of the `ast` Python module, which exports these constants under the same names.

The "`PyCF`" flags above can be combined with "`CO_FUTURE`" flags such as *`CO_FUTURE_ANNOTATIONS`* to enable features normally selectable using future statements. See *Code Object Flags* for a complete list.

CHAPTER 4

참조 횟수

이 섹션의 함수와 매크로는 파이썬 객체의 참조 횟수를 관리하는 데 사용됩니다.

Py_ssize_t **Py_REFCNT**(*PyObject* *o)

Part of the 안정 ABI 버전 *3.14* 이후로. Get the reference count of the Python object *o*.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are *immortal* and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

Use the `Py_SET_REFCNT()` function to set an object reference count.

> **참고**
>
> On *free threaded* builds of Python, returning 1 isn' t sufficient to determine if it' s safe to treat *o* as having no access by other threads. Use `PyUnstable_Object_IsUniquelyReferenced()` for that instead.
>
> See also the function `PyUnstable_Object_IsUniqueReferencedTemporary()`.

버전 3.10에서 변경: `Py_REFCNT()` is changed to the inline static function.

버전 3.11에서 변경: The parameter type is no longer `const PyObject*`.

void **Py_SET_REFCNT**(*PyObject* *o, *Py_ssize_t* refcnt)

Set the object *o* reference counter to *refcnt*.

On Python build with Free Threading, if *refcnt* is larger than `UINT32_MAX`, the object is made *immortal*.

This function has no effect on *immortal* objects.

Added in version 3.9.

버전 3.12에서 변경: Immortal objects are not modified.

void **Py_INCREF**(*PyObject* *o)

Indicate taking a new *strong reference* to object *o*, indicating it is in use and should not be destroyed.

This function has no effect on *immortal* objects.

This function is usually used to convert a *borrowed reference* to a *strong reference* in-place. The `Py_NewRef()` function can be used to create a new *strong reference*.

When done using the object, release is by calling `Py_DECREF()`.

객체는 NULL 일 수 없습니다; NULL이 아닌지 확실하지 않으면, *Py_XINCREF()*를 사용하십시오.

Do not expect this function to actually modify *o* in any way. For at least **some objects**, this function has no effect.

버전 3.12에서 변경: Immortal objects are not modified.

void **Py_XINCREF**(*PyObject* *o)

Similar to *Py_INCREF()*, but the object *o* can be NULL, in which case this has no effect.

See also *Py_XNewRef()*.

PyObject ***Py_NewRef**(*PyObject* *o)

Part of the 안정 ABI 버전 *3.10* 이후로. Create a new *strong reference* to an object: call *Py_INCREF()* on *o* and return the object *o*.

When the *strong reference* is no longer needed, *Py_DECREF()* should be called on it to release the reference.

The object *o* must not be NULL; use *Py_XNewRef()* if *o* can be NULL.

예를 들어:

```
Py_INCREF(obj);
self->attr = obj;
```

can be written as:

```
self->attr = Py_NewRef(obj);
```

See also *Py_INCREF()*.

Added in version 3.10.

PyObject ***Py_XNewRef**(*PyObject* *o)

Part of the 안정 ABI 버전 *3.10* 이후로. Similar to *Py_NewRef()*, but the object *o* can be NULL.

If the object *o* is NULL, the function just returns NULL.

Added in version 3.10.

void **Py_DECREF**(*PyObject* *o)

Release a *strong reference* to object *o*, indicating the reference is no longer used.

This function has no effect on *immortal* objects.

마지막 강한 참조가 해제되면 (즉 객체의 참조 횟수가 0이 되면), 객체 형의 할당 해제 함수 (반드시 NULL이 아니어야 합니다)가 호출됩니다.

This function is usually used to delete a *strong reference* before exiting its scope.

객체는 NULL 일 수 없습니다; NULL이 아닌지 확실하지 않으면, *Py_XDECREF()*를 사용하십시오.

Do not expect this function to actually modify *o* in any way. For at least **some objects**, this function has no effect.

경고

할당 해제 함수는 임의의 파이썬 코드가 호출되도록 할 수 있습니다 (예를 들어, `__del__()` 메서드가 있는 클래스 인스턴스가 할당 해제될 때). 이러한 코드에서의 예외는 전파되지 않지만, 실행된 코드는 모든 파이썬 전역 변수에 자유롭게 액세스할 수 있습니다. 이것은 *Py_DECREF()*가 호출되기 전에 전역 변수에서 도달할 수 있는 모든 객체가 일관성 있는 상태에 있어야 함을 뜻합니다. 예를 들어, 리스트에서 객체를 삭제하는 코드는 삭제된 객체에 대한 참조를 임시 변수에 복사하고, 리스트 데이터 구조를 갱신한 다음, 임시 변수에 대해 *Py_DECREF()*를 호출해야 합니다.

버전 3.12에서 변경: Immortal objects are not modified.

void **Py_XDECREF**(*PyObject* *o)

Similar to `Py_DECREF()`, but the object *o* can be `NULL`, in which case this has no effect. The same warning from `Py_DECREF()` applies here as well.

void **Py_CLEAR**(*PyObject* *o)

객체 *o*에 대한 강한 참조를 해제합니다. 객체는 `NULL` 일 수 있습니다, 이때 매크로는 효과가 없습니다; 그렇지 않으면 인자도 `NULL`로 설정된다는 점을 제외하고는, 효과가 `Py_DECREF()`와 같습니다. 매크로가 임시 변수를 신중하게 사용하고, 참조를 해제하기 전에 인자를 `NULL`로 설정하기 때문에, `Py_DECREF()`에 대한 경고는 전달된 객체와 관련하여 적용되지 않습니다.

가비지 수집 중에 탐색 될 수 있는 객체에 대한 참조를 해제할 때마다 이 매크로를 사용하는 것이 좋습니다.

버전 3.12에서 변경: The macro argument is now only evaluated once. If the argument has side effects, these are no longer duplicated.

void **Py_IncRef**(*PyObject* *o)

Part of the 안정 ABI. Indicate taking a new *strong reference* to object *o*. A function version of `Py_XINCREF()`. It can be used for runtime dynamic embedding of Python.

void **Py_DecRef**(*PyObject* *o)

Part of the 안정 ABI. Release a *strong reference* to object *o*. A function version of `Py_XDECREF()`. It can be used for runtime dynamic embedding of Python.

Py_SETREF(dst, src)

Macro safely releasing a *strong reference* to object *dst* and setting *dst* to *src*.

As in case of `Py_CLEAR()`, "the obvious" code can be deadly:

```
Py_DECREF(dst);
dst = src;
```

The safe way is:

```
Py_SETREF(dst, src);
```

That arranges to set *dst* to *src before* releasing the reference to the old value of *dst*, so that any code triggered as a side-effect of *dst* getting torn down no longer believes *dst* points to a valid object.

Added in version 3.6.

버전 3.12에서 변경: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

Py_XSETREF(dst, src)

Variant of `Py_SETREF` macro that uses `Py_XDECREF()` instead of `Py_DECREF()`.

Added in version 3.6.

버전 3.12에서 변경: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

CHAPTER 5

예외 처리

이 장에서 설명하는 함수를 사용하면 파이썬 예외를 처리하고 발생시킬 수 있습니다. 파이썬 예외 처리의 기본 사항을 이해하는 것이 중요합니다. POSIX `errno` 변수와 비슷하게 작동합니다: 발생한 마지막 에러에 대한 전역 표시기(스레드 당)가 있습니다. 대부분 C API 함수는 성공 시 이를 지우지 않지만, 실패 시 에러의 원인을 나타내도록 설정합니다. 대부분 C API 함수는 에러 표시기도 반환합니다, 일반적으로 포인터를 반환해야 하면 `NULL`, 정수를 반환하면 `-1`을 반환합니다 (예외: `PyArg_*` 함수는 성공하면 `1`을, 실패하면 `0`을 반환합니다).

구체적으로, 에러 표시기는 세 가지 객체 포인터로 구성됩니다: 예외 형, 예외 값 및 트레이스백 객체. 이러한 포인터들은 설정되지 않으면 `NULL`이 될 수 있습니다 (하지만 일부 조합은 금지되어 있습니다, 예를 들어 예외 형이 `NULL`이면 `NULL`이 아닌 트레이스백을 가질 수 없습니다).

호출한 일부 함수가 실패하여 함수가 실패해야 할 때, 일반적으로 에러 표시기를 설정하지 않습니다; 호출된 함수가 이미 설정했습니다. 에러를 처리하고 예외를 지우거나 보유한 모든 리소스(가령 객체 참조나 메모리 할당)를 정리한 후 반환해야 할 책임이 있습니다; 에러를 처리할 준비가 되지 않았을 때 정상적으로 계속되지 않아야 합니다. 에러로 인해 반환하면, 호출자에게 에러가 설정되었음을 알리는 것이 중요합니다. 에러를 처리하지 않거나 신중하게 전파하지 않으면, 파이썬/C API에 대한 추가 호출이 의도한 대로 작동하지 않을 수 있으며 알 수 없는 방식으로 실패할 수 있습니다.

> **참고**
>
> 에러 표시기는 `sys.exc_info()`의 결과가 **아닙니다**. 전자는 아직 포착되지 않은 (따라서 여전히 전파 중인) 예외에 해당하는 반면, 후자는 포착된 후 (따라서 전파가 중단된) 예외를 반환합니다.

5.1 인쇄와 지우기

void **`PyErr_Clear`**()

Part of the 안정 ABI. 에러 표시기를 지웁니다. 에러 표시기가 설정되어 있지 않으면 효과가 없습니다.

void **`PyErr_PrintEx`**(int set_sys_last_vars)

Part of the 안정 ABI. 표준 트레이스백을 `sys.stderr`로 인쇄하고 에러 표시기를 지웁니다. 에러가 `SystemExit`가 **아닌 한**, 이 경우에는 트레이스백이 인쇄되지 않고 파이썬 프로세스는 `SystemExit` 인스턴스에 의해 지정된 에러 코드로 종료됩니다.

에러 표시기가 설정된 경우**에만** 이 함수를 호출하십시오. 그렇지 않으면 치명적인 에러가 발생합니다!

*set_sys_last_vars*가 0이 아니면, 변수 `sys.last_exc` 는 인쇄되는 예외로 설정됩니다. 하위 호환성을 위해, 폐지된 변수 `sys.last_type`, `sys.last_value` 및 `sys.last_traceback` 도 각각 이 예외의 형, 값 및 트레이스백으로 설정됩니다.

버전 3.12에서 변경: The setting of `sys.last_exc` was added.

void **PyErr_Print**()

Part of the 안정 ABI. `PyErr_PrintEx(1)`의 별칭.

void **PyErr_WriteUnraisable**(*PyObject* *obj)

Part of the 안정 ABI. 현재 예외와 *obj* 인자를 사용하여 `sys.unraisablehook()` 을 호출합니다.

이 유틸리티 함수는 예외가 설정되었지만, 인터프리터가 실제로 예외를 발생시킬 수 없을 때 `sys.stderr`에 경고 메시지를 인쇄합니다. 예를 들어, `__del__()` 메서드에서 예외가 발생할 때 사용됩니다.

이 함수는 발생시킬 수 없는 예외가 발생한 문맥을 식별하는 단일 인자 *obj*로 호출됩니다. 가능하면, *obj*의 repr이 경고 메시지에 인쇄됩니다. *obj*가 `NULL`이면, 트래이스백만 인쇄됩니다.

이 함수를 호출할 때 예외를 설정되어 있어야 합니다.

버전 3.4에서 변경: Print a traceback. Print only traceback if *obj* is `NULL`.

버전 3.8에서 변경: Use `sys.unraisablehook()`.

void **PyErr_FormatUnraisable**(const char *format, ...)

Similar to *PyErr_WriteUnraisable()*, but the *format* and subsequent parameters help format the warning message; they have the same meaning and values as in *PyUnicode_FromFormat()*. `PyErr_WriteUnraisable(obj)` is roughly equivalent to `PyErr_FormatUnraisable("Exception ignored in: %R", obj)`. If *format* is `NULL`, only the traceback is printed.

Added in version 3.13.

void **PyErr_DisplayException**(*PyObject* *exc)

Part of the 안정 ABI 버전 *3.12* 이후로. Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

Added in version 3.12.

5.2 예외 발생시키기

이 함수들은 현재 스레드의 에러 표시기를 설정하는 데 도움이 됩니다. 편의를 위해, 이러한 함수 중 일부는 항상 `return` 문에서 사용할 `NULL` 포인터를 반환합니다.

void **PyErr_SetString**(*PyObject* *type, const char *message)

Part of the 안정 ABI. 이것은 에러 표시기를 설정하는 가장 일반적인 방법입니다. 첫 번째 인자는 예외 형을 지정합니다; 일반적으로 표준 예외 중 하나입니다, 예를 들어 *PyExc_RuntimeError*. 새로운 강한 참조를 만들 필요가 없습니다 (예를 들어, *Py_INCREF()*로). 두 번째 인자는 에러 메시지입니다; `'utf-8'` 에서 디코딩됩니다.

void **PyErr_SetObject**(*PyObject* *type, *PyObject* *value)

Part of the 안정 ABI. 이 함수는 *PyErr_SetString()*과 유사하지만, 예외의 "값"에 대해 임의의 파이썬 객체를 지정할 수 있습니다.

PyObject ***PyErr_Format**(*PyObject* *exception, const char *format, ...)

반환값: 항상 *NULL*. *Part of the* 안정 ABI. 이 함수는 에러 표시기를 설정하고 `NULL`을 반환합니다. *exception*은 파이썬 예외 클래스여야 합니다. *format*과 후속 매개 변수는 에러 메시지를 포맷하는 데 도움이 됩니다; *PyUnicode_FromFormat()*에서와 같은 의미와 값을 갖습니다. *format*은 ASCII 인코딩된 문자열입니다.

PyObject ***PyErr_FormatV**(*PyObject* *exception, const char *format, va_list vargs)

반환값: 항상 *NULL*. *Part of the* 안정 ABI 버전 *3.5* 이후로. *PyErr_Format()*과 같지만, 가변 개수의 인자 대신 `va_list` 인자를 취합니다.

Added in version 3.5.

void **PyErr_SetNone**(*PyObject* *type)

Part of the 안정 ABI. 이것은 `PyErr_SetObject(type, Py_None)`의 줄임 표현입니다.

int **PyErr_BadArgument**()

Part of the 안정 ABI. 이것은 `PyErr_SetString(PyExc_TypeError, message)`의 줄임 표현입니다, 여기서 *message*는 잘못된 인자로 내장 연산이 호출되었음을 나타냅니다. 대부분 내부 용입니다.

PyObject ***PyErr_NoMemory**()

반환값: 항상 *NULL*. *Part of the* 안정 ABI. 이것은 `PyErr_SetNone(PyExc_MemoryError)`의 줄임 표현입니다; `NULL`을 반환해서 객체 할당 함수는 메모리가 부족할 때 `return PyErr_NoMemory();` 라고 쓸 수 있습니다.

PyObject ***PyErr_SetFromErrno**(*PyObject* *type)

반환값: 항상 *NULL*. *Part of the* 안정 ABI. C 라이브러리 함수가 에러를 반환하고 C 변수 `errno`를 설정했을 때 예외를 발생시키는 편의 함수입니다. 첫 번째 항목이 정수 `errno` 값이고 두 번째 항목이 (`strerror()`에서 얻은) 해당 에러 메시지인 튜플 객체를 만든 다음, `PyErr_SetObject(type, object)`를 호출합니다. 유닉스에서, `errno` 값이 시스템 호출이 중단되었음을 나타내는 `EINTR`이면, *PyErr_CheckSignals()*를 호출하고 이것이 에러 표시기를 설정하면, 설정된 그대로 둡니다. 이 함수는 항상 `NULL`을 반환해서, 시스템 호출에 대한 래퍼 함수는 시스템 호출이 에러를 반환할 때 `return PyErr_SetFromErrno(type);` 이라고 쓸 수 있습니다.

PyObject ***PyErr_SetFromErrnoWithFilenameObject**(*PyObject* *type, *PyObject* *filenameObject)

반환값: 항상 *NULL*. *Part of the* 안정 ABI. *PyErr_SetFromErrno()*와 유사하지만, *filenameObject*가 `NULL`이 아니면, *type*의 생성자에 세 번째 매개 변수로 전달된다는 추가 동작이 있습니다. `OSError` 예외의 경우, 예외 인스턴스의 `filename` 어트리뷰트를 정의하는 데 사용됩니다.

PyObject ***PyErr_SetFromErrnoWithFilenameObjects**(*PyObject* *type, *PyObject* *filenameObject, *PyObject* *filenameObject2)

반환값: 항상 *NULL*. *Part of the* 안정 ABI 버전 *3.7* 이후로. *PyErr_SetFromErrnoWithFilenameObject()*와 유사하지만, 두 개의 파일명을 취하는 함수가 실패할 때 에러를 발생시키기 위해 두 번째 파일명 객체를 취합니다.

Added in version 3.4.

PyObject ***PyErr_SetFromErrnoWithFilename**(*PyObject* *type, const char *filename)

반환값: 항상 *NULL*. *Part of the* 안정 ABI. *PyErr_SetFromErrnoWithFilenameObject()*와 유사하지만, 파일명이 C 문자열로 제공됩니다. *filename*은 파일시스템 인코딩과 에러 처리기로 디코딩됩니다.

PyObject ***PyErr_SetFromWindowsErr**(int ierr)

반환값: 항상 *NULL*. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. `OSError`를 발생시키는 편의 함수입니다. `0`의 *ierr*로 호출하면, `GetLastError()` 호출에서 반환된 에러 코드가 대신 사용됩니다. Win32 함수 `FormatMessage()`를 호출하여 *ierr*이나 `GetLastError()`가 제공하는 에러 코드의 윈도우 설명을 얻은 다음, `winerror` 어트리뷰트가 에러 코드로 설정되고, `strerror` 어트리뷰트가 (`FormatMessage()`에서 얻은) 해당 에러 메시지로 설정된 `OSError` 객체를 생성한 다음, `PyErr_SetObject(PyExc_OSError, object)`를 호출합니다. 이 함수는 항상 `NULL`을 반환합니다.

가용성: Windows.

PyObject ***PyErr_SetExcFromWindowsErr**(*PyObject* *type, int ierr)

반환값: 항상 *NULL*. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. *PyErr_SetFromWindowsErr()*와 유사하며, 발생시킬 예외 형을 지정하는 추가 매개 변수가 있습니다.

가용성: Windows.

PyObject ***PyErr_SetFromWindowsErrWithFilename**(int ierr, const char *filename)

반환값: 항상 *NULL*. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. *PyErr_SetFromWindowsErr()*과 유사하지만, *filename*이 `NULL`이 아니면, 파일 시스템 인코딩으로 디코딩되고 (`os.fsdecode()`) `OSError`의 생성자에 세 번째 매개 변수로 전달되어 예외 인스턴스의 `filename` 어트리뷰트를 정의하는 데 사용되도록 하는 추가 동작이 있습니다.

가용성: Windows.

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObject**(*PyObject* *type, int ierr, *PyObject* *filename)

반환값: 항상 *NULL*. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. *PyErr_SetExcFromWindowsErr()*과 유사하지만, *filename*이 NULL이 아니면, OSError의 생성자에 세 번째 매개 변수로 전달되어 예외 인스턴스의 filename 어트리뷰트를 정의하는 데 사용되도록 하는 추가 동작이 있습니다.

가용성: Windows.

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObjects**(*PyObject* *type, int ierr, *PyObject* *filename, *PyObject* *filename2)

반환값: 항상 *NULL*. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. *PyErr_SetExcFromWindowsErrWithFilenameObject()*와 유사하지만, 두 번째 파일명 객체를 받아들입니다.

가용성: Windows.

Added in version 3.4.

PyObject ***PyErr_SetExcFromWindowsErrWithFilename**(*PyObject* *type, int ierr, const char *filename)

반환값: 항상 *NULL*. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. *PyErr_SetFromWindowsErrWithFilename()*와 유사하며, 발생시킬 예외 형을 지정하는 추가 매개 변수가 있습니다.

가용성: Windows.

PyObject ***PyErr_SetImportError**(*PyObject* *msg, *PyObject* *name, *PyObject* *path)

반환값: 항상 *NULL*. *Part of the* 안정 ABI 버전 *3.7* 이후로. ImportError를 발생시키는 편의 함수입니다. *msg*는 예외의 메시지 문자열로 설정됩니다. 둘 다 NULL이 될 수 있는, *name*과 *path*는 각각 ImportError의 name과 path 어트리뷰트로 설정됩니다.

Added in version 3.3.

PyObject ***PyErr_SetImportErrorSubclass**(*PyObject* *exception, *PyObject* *msg, *PyObject* *name, *PyObject* *path)

반환값: 항상 *NULL*. *Part of the* 안정 ABI 버전 *3.6* 이후로. *PyErr_SetImportError()* 와 매우 비슷하지만, 이 함수는 발생시킬 ImportError의 서브 클래스를 지정할 수 있습니다.

Added in version 3.6.

void **PyErr_SyntaxLocationObject**(*PyObject* *filename, int lineno, int col_offset)

현재 예외에 대한 파일(file), 줄(line) 및 오프셋(offset) 정보를 설정합니다. 현재 예외가 SyntaxError가 아니면, 추가 어트리뷰트를 설정하여, 예외 인쇄 하위 시스템이 예외가 SyntaxError라고 생각하게 합니다.

Added in version 3.4.

void **PyErr_SyntaxLocationEx**(const char *filename, int lineno, int col_offset)

Part of the 안정 ABI 버전 *3.7* 이후로. *PyErr_SyntaxLocationObject()* 와 비슷하지만, *filename*은 파일시스템 인코딩과 에러 처리기로 디코딩되는 바이트 문자열입니다.

Added in version 3.2.

void **PyErr_SyntaxLocation**(const char *filename, int lineno)

Part of the 안정 ABI. *PyErr_SyntaxLocationEx()* 와 비슷하지만, *col_offset* 매개 변수는 생략됩니다.

void **PyErr_BadInternalCall**()

Part of the 안정 ABI. 이것은 PyErr_SetString(PyExc_SystemError, message)의 줄임 표현입니다. 여기서 *message*는 내부 연산(예를 들어 파이썬/C API 함수)이 잘못된 인자로 호출되었음을 나타냅니다. 대부분 내부 용입니다.

5.3 경고 발행하기

이 함수를 사용하여 C 코드에서 경고를 발행하십시오. 파이썬 `warnings` 모듈에서 내보낸 유사한 함수를 미러링합니다. 일반적으로 *sys.stderr*에 경고 메시지를 인쇄합니다; 그러나, 사용자가 경고를 에러로 전환하도록 지정했을 수도 있으며, 이 경우 예외가 발생합니다. 경고 장치의 문제로 인해 이 함수가 예외를 발생시키는 것도 가능합니다. 예외가 발생하지 않으면 반환 값은 `0`이고, 예외가 발생하면 `-1`입니다. (경고 메시지가 실제로 인쇄되는지나 예외의 이유를 확인할 수 없습니다; 이것은 의도적입니다.) 예외가 발생하면, 호출자는 정상적인 예외 처리를 수행해야 합니다 (예를 들어, 소유한 참조를 `Py_DECREF()` 하고 에러값을 반환합니다).

int **PyErr_WarnEx**(*PyObject* *category, const char *message, *Py_ssize_t* stack_level)

Part of the 안정 ABI. 경고 메시지를 발행합니다. *category* 인자는 경고 범주(아래를 참조하십시오)나 `NULL`입니다; *message* 인자는 UTF-8로 인코딩된 문자열입니다. *stack_level*은 스택 프레임 수를 제공하는 양수입니다; 해당 스택 프레임에서 현재 실행 중인 코드 줄에서 경고가 발생합니다. *stack_level*이 1이면 `PyErr_WarnEx()`를 호출하는 함수, 2는 그 위의 함수, 등등.

경고 범주는 `PyExc_Warning`의 서브 클래스여야 합니다. `PyExc_Warning`은 `PyExc_Exception`의 서브 클래스입니다; 기본 경고 범주는 `PyExc_RuntimeWarning`입니다. 표준 파이썬 경고 범주는 이름이 *Warning types*에 열거된 전역 변수로 제공됩니다.

경고 제어에 대한 자세한 내용은, `warnings` 모듈 설명서와 명령 줄 설명서에서 `-W` 옵션을 참조하십시오. 경고 제어를 위한 C API는 없습니다.

int **PyErr_WarnExplicitObject**(*PyObject* *category, *PyObject* *message, *PyObject* *filename, int lineno, *PyObject* *module, *PyObject* *registry)

모든 경고 어트리뷰트를 명시적으로 제어하면서 경고 메시지를 발행합니다. 이것은 파이썬 함수 `warnings.warn_explicit()`에 대한 간단한 래퍼입니다; 자세한 내용은 거기를 참조하십시오. 그곳에 설명된 기본 효과를 얻으려면 *module*과 *registry* 인자를 `NULL`로 설정해야 합니다.

Added in version 3.4.

int **PyErr_WarnExplicit**(*PyObject* *category, const char *message, const char *filename, int lineno, const char *module, *PyObject* *registry)

Part of the 안정 ABI. *message*와 *module*이 UTF-8 인코딩된 문자열이고, *filename*은 파일시스템 인코딩과 에러 처리기로 디코딩된다는 점을 제외하면 `PyErr_WarnExplicitObject()` 와 유사합니다.

int **PyErr_WarnFormat**(*PyObject* *category, *Py_ssize_t* stack_level, const char *format, ...)

Part of the 안정 ABI. `PyErr_WarnEx()`와 유사한 함수지만, `PyUnicode_FromFormat()`을 사용하여 경고 메시지를 포맷합니다. *format*은 ASCII 인코딩된 문자열입니다.

Added in version 3.2.

int **PyErr_ResourceWarning**(*PyObject* *source, *Py_ssize_t* stack_level, const char *format, ...)

Part of the 안정 ABI 버전 *3.6* 이후로. `PyErr_WarnFormat()`과 유사한 함수지만, *category*는 `ResourceWarning`이고 *source*를 `warnings.WarningMessage`로 전달합니다.

Added in version 3.6.

5.4 에러 표시기 조회하기

PyObject ***PyErr_Occurred**()

반환값: 빌린 참조. *Part of the* 안정 ABI. 에러 표시기가 설정되었는지 테스트합니다. 설정되었으면, 예외 *type*(`PyErr_Set*` 함수나 `PyErr_Restore()`에 대한 마지막 호출의 첫 번째 인자)을 반환합니다. 설정되지 않았으면, `NULL`을 반환합니다. 여러분이 반환 값에 대한 참조를 소유하지 않아서, `Py_DECREF()` 할 필요가 없습니다.

The caller must have an *attached thread state*.

> **참고**
>
> 반환 값을 특정 예외와 비교하지 마십시오; 대신 *PyErr_ExceptionMatches()*를 사용하십시오, 아래를 참조하십시오. (클래스 예외의 경우 예외가 클래스 대신 인스턴스이거나, 예상하는 예외의 서브 클래스일 수 있어서 비교는 실패하기 쉽습니다.)

int **PyErr_ExceptionMatches**(*PyObject* *exc)

Part of the 안정 ABI. `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`와 동등합니다. 예외가 실제로 설정되었을 때만 호출해야 합니다; 예외가 발생하지 않았으면 메모리 액세스 위반이 발생합니다.

int **PyErr_GivenExceptionMatches**(*PyObject* *given, *PyObject* *exc)

Part of the 안정 ABI. *given* 예외가 *exc*의 예외 형과 일치하면 참을 반환합니다. *exc*가 클래스 객체이면, *given*이 서브 클래스의 인스턴스일 때도 참을 반환합니다. *exc*가 튜플이면, 튜플에 있는 모든 예외 형(그리고 서브 튜플도 재귀적으로)을 일치를 위해 검색합니다.

PyObject ***PyErr_GetRaisedException**(void)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.12* 이후로. Return the exception currently being raised, clearing the error indicator at the same time. Return `NULL` if the error indicator is not set.

이 함수는 예외를 포착해야 하는 코드나 에러 표시기를 일시적으로 저장하고 복원해야 하는 코드에서 사용됩니다.

예를 들어:

```
{
   PyObject *exc = PyErr_GetRaisedException();

   /* ... 다른 에러를 생성할 수 있는 코드 ... */

   PyErr_SetRaisedException(exc);
}
```

> **더 보기**
>
> *PyErr_GetHandledException()*, to save the exception currently being handled.

Added in version 3.12.

void **PyErr_SetRaisedException**(*PyObject* *exc)

Part of the 안정 ABI 버전 *3.12* 이후로. Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

> **경고**
>
> This call steals a reference to *exc*, which must be a valid exception.

Added in version 3.12.

void **PyErr_Fetch**(*PyObject* **ptype, *PyObject* **pvalue, *PyObject* **ptraceback)

Part of the 안정 ABI. 버전 3.12부터 폐지됨: Use *PyErr_GetRaisedException()* instead.

주소가 전달된 세 개의 변수로 에러 표시기를 꺼냅니다. 에러 표시기가 설정되지 않았으면, 세 변수를 모두 `NULL`로 설정합니다. 설정되었으면, 지워지고 꺼낸 각 객체에 대한 참조를 여러분이 소유합니다. 값과 트레이스백 객체는 형 객체가 그렇지 않을 때도 `NULL`일 수 있습니다.

> **참고**
>
> 이 함수는 일반적으로 예외를 포착해야 하거나 에러 표시기를 일시적으로 저장하고 복원해야 하는 레거시 코드에서만 사용됩니다.
>
> 예를 들어:

```
{
   PyObject *type, *value, *traceback;
   PyErr_Fetch(&type, &value, &traceback);

   /* ... 다른 에러를 생성할 수 있는 코드 ... */

   PyErr_Restore(type, value, traceback);
}
```

void **PyErr_Restore**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)

Part of the 안정 ABI. 버전 3.12부터 폐지됨: Use `PyErr_SetRaisedException()` instead.

세 객체, *type*, *value* 및 *traceback*으로 에러 표시기를 설정합니다. 이미 설정되어 있으면, 기존 예외를 지웁니다. 객체가 `NULL`이면, 에러 표시기가 지워집니다. `NULL` type과 함께 `NULL`이 아닌 value나 traceback 을 전달하지 마십시오. 예외 형은 클래스여야 합니다. 잘못된 예외 형이나 값을 전달하지 마십시오. (이러한 규칙을 위반하면 나중에 미묘한 문제가 발생합니다.) 이 호출은 각 객체에 대한 참조를 제거합니다: 호출 전에 각 객체에 대한 참조를 소유해야 하며 호출 후에는 더는 이러한 참조를 소유하지 않습니다. (이것을 이해할 수 없다면, 이 함수를 사용하지 마십시오. 경고했습니다.)

> **참고**
>
> 이 함수는 일반적으로 에러 표시기를 일시적으로 저장하고 복원해야 하는 레거시 코드에서만 사용됩니다. 현재 에러 표시기를 저장하려면 `PyErr_Fetch()`를 사용하십시오.

void **PyErr_NormalizeException**(*PyObject* **exc, *PyObject* **val, *PyObject* **tb)

Part of the 안정 ABI. 버전 3.12부터 폐지됨: Use `PyErr_GetRaisedException()` instead, to avoid any possible de-normalization.

특정 상황에서, 아래의 `PyErr_Fetch()`가 반환하는 값은 "비 정규화"되었을 수 있습니다. 즉, `*exc`는 클래스 객체이지만 `*val`은 같은 클래스의 인스턴스가 아닙니다. 이 함수는 이 경우 클래스를 인스턴스 화하는 데 사용할 수 있습니다. 값이 이미 정규화되어 있으면, 아무 일도 일어나지 않습니다. 지연된 정규화는 성능 향상을 위해 구현됩니다.

> **참고**
>
> 이 함수 예외 값에 `__traceback__` 어트리뷰트를 묵시적으로 설정하지 않습니다. 트레이스백을 적절하게 설정해야 하면, 다음과 같은 추가 스니펫이 필요합니다:

```
if (tb != NULL) {
  PyException_SetTraceback(val, tb);
}
```

PyObject ***PyErr_GetHandledException**(void)

Part of the 안정 ABI 버전 *3.11* 이후로. `sys.exception()`이 반환하는 것과 같은, 활성 예외 인스턴스를 꺼냅니다. 이것은 새로 발생한 예외가 아니라, 이미 포착된 예외를 가리킵니다. 예외에 대한 새 참조나 `NULL`을 반환합니다. 인터프리터의 예외 상태를 수정하지 않습니다.

> **참고**
>
> 이 함수는 일반적으로 예외를 처리하려는 코드에서 사용되지 않습니다. 오히려, 코드가 예외 상태를 임시로 저장하고 복원해야 할 때 사용할 수 있습니다. 예외 상태를 복원하거나 지우려면 *PyErr_SetHandledException()*를 사용하십시오.

Added in version 3.11.

void **PyErr_SetHandledException**(*PyObject* *exc)

Part of the 안정 ABI 버전 *3.11* 이후로. `sys.exception()`으로 알려진 것과 같은, 활성 예외를 설정합니다. 이것은 새로 발생한 예외가 아니라, 이미 포착된 예외를 가리킵니다. 예외 상태를 지우려면, `NULL`을 전달하십시오.

> **참고**
>
> 이 함수는 일반적으로 예외를 처리하려는 코드에서 사용되지 않습니다. 오히려, 코드가 예외 상태를 임시로 저장하고 복원해야 할 때 사용할 수 있습니다. 예외 상태를 얻으려면 *PyErr_GetHandledException()*를 사용하십시오.

Added in version 3.11.

void **PyErr_GetExcInfo**(*PyObject* **ptype, *PyObject* **pvalue, *PyObject* **ptraceback)

Part of the 안정 ABI 버전 *3.7* 이후로. `sys.exc_info()`로 알려진 것과 같은, 예외 정보의 구식 표현을 꺼냅니다. 이것은 새로 발생한 예외가 아니라, 이미 포착된 예외를 가리킵니다. 세 객체에 대한 새 참조를 반환합니다. 이 중 어느 것이든 `NULL`일 수 있습니다. 예외 정보 상태를 수정하지 않습니다. 이 함수는 하위 호환성을 위해 유지됩니다. 대신 *PyErr_GetHandledException()*을 사용하세요.

> **참고**
>
> 이 함수는 일반적으로 예외를 처리하려는 코드에서 사용되지 않습니다. 오히려, 코드가 예외 상태를 임시로 저장하고 복원해야 할 때 사용할 수 있습니다. 예외 상태를 복원하거나 지우려면 *PyErr_SetExcInfo()*를 사용하십시오.

Added in version 3.3.

void **PyErr_SetExcInfo**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)

Part of the 안정 ABI 버전 *3.7* 이후로. `sys.exc_info()`로 알려진 것과 같은, 예외 정보를 설정합니다. 이것은 새로 발생한 예외가 아니라, 이미 포착된 예외를 가리킵니다. 이 함수는 인자의 참조를 훔칩니다. 예외 상태를 지우려면, 세 인자 모두에 `NULL`을 전달하십시오. 이 함수는 하위 호환성을 위해 유지됩니다. 대신 *PyErr_SetHandledException()*을 사용하십시오.

> **참고**
>
> 이 함수는 일반적으로 예외를 처리하려는 코드에서 사용되지 않습니다. 오히려, 코드가 예외 상태를 임시로 저장하고 복원해야 할 때 사용할 수 있습니다. 예외 상태를 읽으려면 *PyErr_GetExcInfo()*를 사용하십시오.

Added in version 3.3.

버전 3.11에서 변경: The `type` and `traceback` arguments are no longer used and can be NULL. The interpreter now derives them from the exception instance (the `value` argument). The function still steals references of all three arguments.

5.5 시그널 처리하기

int **PyErr_CheckSignals**()

Part of the 안정 ABI. This function interacts with Python' s signal handling.

If the function is called from the main thread and under the main Python interpreter, it checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the `signal` module is supported, this can invoke a signal handler written in Python.

The function attempts to handle all pending signals, and then returns `0`. However, if a Python signal handler raises an exception, the error indicator is set and the function returns `-1` immediately (such that other pending signals may not have been handled yet: they will be on the next *PyErr_CheckSignals()* invocation).

If the function is called from a non-main thread, or under a non-main Python interpreter, it does nothing and returns `0`.

This function can be called by long-running C code that wants to be interruptible by user requests (such as by pressing Ctrl-C).

> **참고**
>
> The default Python signal handler for `SIGINT` raises the `KeyboardInterrupt` exception.

void **PyErr_SetInterrupt**()

Part of the 안정 ABI. Simulate the effect of a `SIGINT` signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

> **참고**
>
> This function is async-signal-safe. It can be called without an *attached thread state* and from a C signal handler.

int **PyErr_SetInterruptEx**(int signum)

Part of the 안정 ABI 버전 *3.10* 이후로. 시그널 도착의 효과를 시뮬레이션합니다. 다음에 *PyErr_CheckSignals()*가 호출되면, 주어진 시그널 번호에 대한 파이썬 시그널 처리기가 호출됩니다.

This function can be called by C code that sets up its own signal handling and wants Python signal handlers to be invoked as expected when an interruption is requested (for example when the user presses Ctrl-C to interrupt an operation).

주어진 시그널이 파이썬에서 처리되지 않으면 (`signal.SIG_DFL`이나 `signal.SIG_IGN`으로 설정되어서), 무시됩니다.

If *signum* is outside of the allowed range of signal numbers, `-1` is returned. Otherwise, `0` is returned. The error indicator is never changed by this function.

> **참고**
>
> This function is async-signal-safe. It can be called without an *attached thread state* and from a C signal handler.

Added in version 3.10.

int **PySignal_SetWakeupFd**(int fd)

이 유틸리티 함수는 시그널이 수신될 때마다 시그널 번호가 단일 바이트로 기록되는 파일 기술자를 지정합니다. *fd*는 비 블로킹이어야 합니다. 이전의 파일 기술자를 반환합니다.

값 -1은 기능을 비활성화합니다; 이것이 초기 상태입니다. 이것은 파이썬의 signal.set_wakeup_fd()와 동등하지만, 에러 검사는 없습니다. *fd*는 유효한 파일 기술자여야 합니다. 함수는 메인 스레드에서만 호출되어야 합니다.

버전 3.5에서 변경: 윈도우에서, 함수는 이제 소켓 핸들도 지원합니다.

5.6 예외 클래스

PyObject ***PyErr_NewException**(const char *name, *PyObject* *base, *PyObject* *dict)

반환값: 새 참조. *Part of the* 안정 ABI. 이 유틸리티 함수는 새 예외 클래스를 만들고 반환합니다. *name* 인자는 새 예외의 이름, module.classname 형식의 C 문자열이어야 합니다. *base*와 *dict* 인자는 일반적으로 NULL입니다. 이렇게 하면 Exception(C에서 *PyExc_Exception*으로 액세스할 수 있습니다)에서 파생된 클래스 객체가 만들어집니다.

새 클래스의 __module__ 어트리뷰트는 *name* 인자의 첫 번째 부분(마지막 점까지)으로 설정되고, 클래스 이름은 마지막 부분(마지막 점 뒤)으로 설정됩니다. *base* 인자는 대체 베이스 클래스를 지정하는 데 사용할 수 있습니다; 하나의 클래스나 클래스의 튜플일 수 있습니다. *dict* 인자는 클래스 변수와 메서드의 딕셔너리를 지정하는 데 사용할 수 있습니다.

PyObject ***PyErr_NewExceptionWithDoc**(const char *name, const char *doc, *PyObject* *base, *PyObject* *dict)

반환값: 새 참조. *Part of the* 안정 ABI. 새로운 예외 클래스에 독스트링을 쉽게 부여할 수 있다는 점을 제외하면 *PyErr_NewException()*과 같습니다: *doc*이 NULL이 아니면, 예외 클래스에 대한 독스트링으로 사용됩니다.

Added in version 3.2.

int **PyExceptionClass_Check**(*PyObject* *ob)

Return non-zero if *ob* is an exception class, zero otherwise. This function always succeeds.

const char ***PyExceptionClass_Name**(*PyObject* *ob)

Part of the 안정 ABI 버전 *3.8* 이후로. Return *tp_name* of the exception class *ob*.

5.7 예외 객체

PyObject ***PyException_GetTraceback**(*PyObject* *ex)

반환값: 새 참조. *Part of the* 안정 ABI. 파이썬에서 __traceback__ 어트리뷰트를 통해 액세스 할 수 있는 새로운 참조로 예외와 관련된 트레이스백을 반환합니다. 관련된 트레이스백이 없으면, NULL을 반환합니다.

int **PyException_SetTraceback**(*PyObject* *ex, *PyObject* *tb)

Part of the 안정 ABI. 예외와 관련된 트레이스백을 *tb*로 설정합니다. 지우려면 Py_None을 사용하십시오.

PyObject ***PyException_GetContext**(*PyObject* *ex)

반환값: 새 참조. *Part of the* 안정 ABI. 파이썬에서 __context__ 어트리뷰트를 통해 액세스 할 수 있는 새 참조로 예외와 연관된 컨텍스트(다른 예외 인스턴스, 이것을 처리하는 도중 *ex*가 발생했습니다)를 반환합니다. 연결된 컨텍스트가 없으면 NULL을 반환합니다.

void **PyException_SetContext**(*PyObject* *ex, *PyObject* *ctx)

Part of the 안정 ABI. 예외와 연관된 컨텍스트를 *ctx*로 설정합니다. 지우려면 NULL을 사용하십시오. *ctx*가 예외 인스턴스인지 확인하는 형 검사는 없습니다. 이것은 *ctx*에 대한 참조를 훔칩니다.

PyObject ***PyException_GetCause**(*PyObject* *ex)

반환값: 새 참조. *Part of the* 안정 ABI. 파이썬에서 __cause__ 어트리뷰트를 통해 액세스 할 수 있는 새 참조로 예외와 관련된 원인(예외 인스턴스나 None, raise ... from ...으로 설정됩니다)을 반환합니다.

void **PyException_SetCause**(*PyObject* *ex, *PyObject* *cause)

Part of the 안정 ABI. 예외와 관련된 원인을 *cause*로 설정합니다. 지우려면 `NULL`을 사용하십시오. *cause*가 예외 인스턴스나 `None`인지 확인하는 형 검사는 없습니다. 이것은 *cause*에 대한 참조를 훔칩니다.

`__suppress_context__` 어트리뷰트는 이 함수에 의해 묵시적으로 `True`로 설정됩니다.

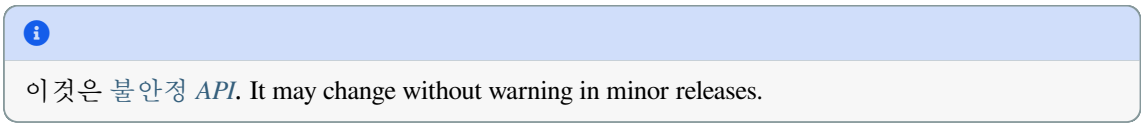
PyObject ***PyException_GetArgs**(*PyObject* *ex)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.12* 이후로. Return `args` of exception *ex*.

void **PyException_SetArgs**(*PyObject* *ex, *PyObject* *args)

Part of the 안정 ABI 버전 *3.12* 이후로. Set `args` of exception *ex* to *args*.

PyObject ***PyUnstable_Exc_PrepReraiseStar**(*PyObject* *orig, *PyObject* *excs)



이것은 불안정 *API*. It may change without warning in minor releases.

Implement part of the interpreter' s implementation of `except*`. *orig* is the original exception that was caught, and *excs* is the list of the exceptions that need to be raised. This list contains the unhandled part of *orig*, if any, as well as the exceptions that were raised from the `except*` clauses (so they have a different traceback from *orig*) and those that were reraised (and have the same traceback as *orig*). Return the `ExceptionGroup` that needs to be reraised in the end, or `None` if there is nothing to reraise.

Added in version 3.12.

5.8 유니코드 예외 객체

다음 함수는 C에서 유니코드 예외를 만들고 수정하는 데 사용됩니다.

PyObject ***PyUnicodeDecodeError_Create**(const char *encoding, const char *object, *Py_ssize_t* length, *Py_ssize_t* start, *Py_ssize_t* end, const char *reason)

반환값: 새 참조. *Part of the* 안정 ABI. *encoding*, *object*, *length*, *start*, *end* 및 *reason* 어트리뷰트를 사용하여 `UnicodeDecodeError` 객체를 만듭니다. *encoding*과 *reason*은 UTF-8로 인코딩된 문자열입니다.

PyObject ***PyUnicodeDecodeError_GetEncoding**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetEncoding**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 예외 객체의 *encoding* 어트리뷰트를 반환합니다.

PyObject ***PyUnicodeDecodeError_GetObject**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetObject**(*PyObject* *exc)

PyObject ***PyUnicodeTranslateError_GetObject**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 예외 객체의 *object* 어트리뷰트를 반환합니다.

int **PyUnicodeDecodeError_GetStart**(*PyObject* *exc, *Py_ssize_t* *start)

int **PyUnicodeEncodeError_GetStart**(*PyObject* *exc, *Py_ssize_t* *start)

int **PyUnicodeTranslateError_GetStart**(*PyObject* *exc, *Py_ssize_t* *start)

Part of the 안정 ABI. 주어진 예외 객체의 *start* 어트리뷰트를 가져와서 **start*에 배치합니다. *start*는 `NULL`이 아니어야 합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

If the `UnicodeError.object` is an empty sequence, the resulting *start* is `0`. Otherwise, it is clipped to `[0, len(object) - 1]`.

더 보기

`UnicodeError.start`

int **PyUnicodeDecodeError_SetStart**(*PyObject* *exc, *Py_ssize_t* start)

int **PyUnicodeEncodeError_SetStart**(*PyObject* *exc, *Py_ssize_t* start)

int **PyUnicodeTranslateError_SetStart**(*PyObject* *exc, *Py_ssize_t* start)

Part of the 안정 ABI. Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

> **참고**
>
> While passing a negative *start* does not raise an exception, the corresponding getters will not consider it as a relative offset.

int **PyUnicodeDecodeError_GetEnd**(*PyObject* *exc, *Py_ssize_t* *end)

int **PyUnicodeEncodeError_GetEnd**(*PyObject* *exc, *Py_ssize_t* *end)

int **PyUnicodeTranslateError_GetEnd**(*PyObject* *exc, *Py_ssize_t* *end)

Part of the 안정 ABI. 주어진 예외 객체의 *end* 어트리뷰트를 가져와서 **end*에 배치합니다. *end*는 `NULL`이 아니어야 합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

If the `UnicodeError.object` is an empty sequence, the resulting *end* is `0`. Otherwise, it is clipped to `[1, len(object)]`.

int **PyUnicodeDecodeError_SetEnd**(*PyObject* *exc, *Py_ssize_t* end)

int **PyUnicodeEncodeError_SetEnd**(*PyObject* *exc, *Py_ssize_t* end)

int **PyUnicodeTranslateError_SetEnd**(*PyObject* *exc, *Py_ssize_t* end)

Part of the 안정 ABI. 주어진 예외 객체의 *end* 어트리뷰트를 *end*로 설정합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

> **더 보기**
>
> `UnicodeError.end`

PyObject ***PyUnicodeDecodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeTranslateError_GetReason**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 예외 객체의 *reason* 어트리뷰트를 반환합니다.

int **PyUnicodeDecodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeEncodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeTranslateError_SetReason**(*PyObject* *exc, const char *reason)

Part of the 안정 ABI. 주어진 예외 객체의 *reason* 어트리뷰트를 *reason*으로 설정합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

5.9 재귀 제어

이 두 함수는 코어와 확장 모듈 모두에서 C 수준에서 안전한 재귀 호출을 수행하는 방법을 제공합니다. 재귀 코드가 반드시 파이썬 코드를 호출하지 않는 경우 필요합니다 (파이썬 코드는 재귀 깊이를 자동으로 추적합니다). 호출 프로토콜이 재귀 처리를 처리하기 때문에 *tp_call* 구현에도 필요하지 않습니다.

int **Py_EnterRecursiveCall**(const char *where)

Part of the 안정 ABI 버전 *3.9* 이후로. 재귀적 C 수준 호출이 막 수행되려고 하는 지점을 표시합니다.

The function then checks if the stack limit is reached. If this is the case, a `RecursionError` is set and a nonzero value is returned. Otherwise, zero is returned.

*where*는 재귀 깊이 제한으로 인한 `RecursionError` 메시지에 이어붙일 `" in instance check"`와 같은 UTF-8 인코딩된 문자열이어야 합니다.

버전 3.9에서 변경: 이 함수는 이제 제한된 *API*에서도 사용할 수 있습니다.

void **Py_LeaveRecursiveCall**(void)

Part of the 안정 ABI 버전 *3.9* 이후로. `Py_EnterRecursiveCall()`을 종료합니다. `Py_EnterRecursiveCall()`의 각 성공적인 호출마다 한 번씩 호출되어야 합니다.

버전 3.9에서 변경: 이 함수는 이제 제한된 *API*에서도 사용할 수 있습니다.

컨테이너형에 대해 `tp_repr`을 올바르게 구현하려면 특별한 재귀 처리가 필요합니다. 스택을 보호하는 것 외에도, `tp_repr`은 순환을 방지하기 위해 객체를 추적해야 합니다. 다음 두 함수는 이 기능을 쉽게 만듭니다. 사실상, 이들은 `reprlib.recursive_repr()`에 대한 C 동등물입니다.

int **Py_ReprEnter**(*PyObject* *object)

Part of the 안정 ABI. 순환을 감지하기 위해 `tp_repr` 구현 시작 시 호출됩니다.

객체가 이미 처리되었으면, 함수는 양의 정수를 반환합니다. 이 경우 `tp_repr` 구현은 순환을 나타내는 문자열 객체를 반환해야 합니다. 예를 들어, `dict` 객체는 `{...}`를 반환하고 `list` 객체는 `[...]`를 반환합니다.

재귀 제한에 도달하면 함수는 음의 정수를 반환합니다. 이 경우 `tp_repr` 구현은 일반적으로 `NULL`을 반환해야 합니다.

그렇지 않으면, 함수는 0을 반환하고 `tp_repr` 구현은 정상적으로 계속될 수 있습니다.

void **Py_ReprLeave**(*PyObject* *object)

Part of the 안정 ABI. `Py_ReprEnter()`를 종료합니다. 0을 반환하는 `Py_ReprEnter()` 호출마다 한 번씩 호출해야 합니다.

5.10 Exception and warning types

All standard Python exceptions and warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects.

For completeness, here are all the variables:

5.10.1 Exception types

C name	Python name
PyObject ***PyExc_BaseException** *Part of the* 안정 ABI.	`BaseException`
PyObject ***PyExc_BaseExceptionGroup** *Part of the* 안정 ABI 버전 *3.11* 이후로.	`BaseExceptionGroup`
PyObject ***PyExc_Exception** *Part of the* 안정 ABI.	`Exception`
PyObject ***PyExc_ArithmeticError** *Part of the* 안정 ABI.	`ArithmeticError`
PyObject ***PyExc_AssertionError** *Part of the* 안정 ABI.	`AssertionError`

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

C name	Python name
PyObject ***PyExc_AttributeError** *Part of the* 안정 ABI.	`AttributeError`
PyObject ***PyExc_BlockingIOError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`BlockingIOError`
PyObject ***PyExc_BrokenPipeError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`BrokenPipeError`
PyObject ***PyExc_BufferError** *Part of the* 안정 ABI.	`BufferError`
PyObject ***PyExc_ChildProcessError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`ChildProcessError`
PyObject ***PyExc_ConnectionAbortedError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`ConnectionAbortedError`
PyObject ***PyExc_ConnectionError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`ConnectionError`
PyObject ***PyExc_ConnectionRefusedError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`ConnectionRefusedError`
PyObject ***PyExc_ConnectionResetError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`ConnectionResetError`
PyObject ***PyExc_EOFError** *Part of the* 안정 ABI.	`EOFError`
PyObject ***PyExc_FileExistsError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`FileExistsError`
PyObject ***PyExc_FileNotFoundError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`FileNotFoundError`
PyObject ***PyExc_FloatingPointError** *Part of the* 안정 ABI.	`FloatingPointError`

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

C name	Python name
PyObject ***PyExc_GeneratorExit** *Part of the* 안정 ABI.	`GeneratorExit`
PyObject ***PyExc_ImportError** *Part of the* 안정 ABI.	`ImportError`
PyObject ***PyExc_IndentationError** *Part of the* 안정 ABI.	`IndentationError`
PyObject ***PyExc_IndexError** *Part of the* 안정 ABI.	`IndexError`
PyObject ***PyExc_InterruptedError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`InterruptedError`
PyObject ***PyExc_IsADirectoryError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`IsADirectoryError`
PyObject ***PyExc_KeyError** *Part of the* 안정 ABI.	`KeyError`
PyObject ***PyExc_KeyboardInterrupt** *Part of the* 안정 ABI.	`KeyboardInterrupt`
PyObject ***PyExc_LookupError** *Part of the* 안정 ABI.	`LookupError`
PyObject ***PyExc_MemoryError** *Part of the* 안정 ABI.	`MemoryError`
PyObject ***PyExc_ModuleNotFoundError** *Part of the* 안정 ABI 버전 *3.6* 이후로.	`ModuleNotFoundError`
PyObject ***PyExc_NameError** *Part of the* 안정 ABI.	`NameError`
PyObject ***PyExc_NotADirectoryError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`NotADirectoryError`

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

C name	Python name
PyObject ***PyExc_NotImplementedError** *Part of the* 안정 ABI.	`NotImplementedError`
PyObject ***PyExc_OSError** *Part of the* 안정 ABI.	`OSError`
PyObject ***PyExc_OverflowError** *Part of the* 안정 ABI.	`OverflowError`
PyObject ***PyExc_PermissionError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`PermissionError`
PyObject ***PyExc_ProcessLookupError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`ProcessLookupError`
PyObject ***PyExc_PythonFinalizationError**	`PythonFinalizationError`
PyObject ***PyExc_RecursionError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`RecursionError`
PyObject ***PyExc_ReferenceError** *Part of the* 안정 ABI.	`ReferenceError`
PyObject ***PyExc_RuntimeError** *Part of the* 안정 ABI.	`RuntimeError`
PyObject ***PyExc_StopAsyncIteration** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`StopAsyncIteration`
PyObject ***PyExc_StopIteration** *Part of the* 안정 ABI.	`StopIteration`
PyObject ***PyExc_SyntaxError** *Part of the* 안정 ABI.	`SyntaxError`
PyObject ***PyExc_SystemError** *Part of the* 안정 ABI.	`SystemError`
PyObject ***PyExc_SystemExit** *Part of the* 안정 ABI.	`SystemExit`

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

C name	Python name
PyObject ***PyExc_TabError** *Part of the* 안정 ABI.	`TabError`
PyObject ***PyExc_TimeoutError** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`TimeoutError`
PyObject ***PyExc_TypeError** *Part of the* 안정 ABI.	`TypeError`
PyObject ***PyExc_UnboundLocalError** *Part of the* 안정 ABI.	`UnboundLocalError`
PyObject ***PyExc_UnicodeDecodeError** *Part of the* 안정 ABI.	`UnicodeDecodeError`
PyObject ***PyExc_UnicodeEncodeError** *Part of the* 안정 ABI.	`UnicodeEncodeError`
PyObject ***PyExc_UnicodeError** *Part of the* 안정 ABI.	`UnicodeError`
PyObject ***PyExc_UnicodeTranslateError** *Part of the* 안정 ABI.	`UnicodeTranslateError`
PyObject ***PyExc_ValueError** *Part of the* 안정 ABI.	`ValueError`
PyObject ***PyExc_ZeroDivisionError** *Part of the* 안정 ABI.	`ZeroDivisionError`

Added in version 3.3: *PyExc_BlockingIOError*, *PyExc_BrokenPipeError*, *PyExc_ChildProcessError*, *PyExc_ConnectionError*, *PyExc_ConnectionAbortedError*, *PyExc_ConnectionRefusedError*, *PyExc_ConnectionResetError*, *PyExc_FileExistsError*, *PyExc_FileNotFoundError*, *PyExc_InterruptedError*, *PyExc_IsADirectoryError*, *PyExc_NotADirectoryError*, *PyExc_PermissionError*, *PyExc_ProcessLookupError* 및 *PyExc_TimeoutError*는 **PEP 3151**을 따라 도입되었습니다.

Added in version 3.5: *PyExc_StopAsyncIteration* 과 *PyExc_RecursionError*.

Added in version 3.6: *PyExc_ModuleNotFoundError*.

Added in version 3.11: *PyExc_BaseExceptionGroup*.

5.10.2 OSError aliases

The following are a compatibility aliases to *PyExc_OSError*.

버전 3.3에서 변경: 이러한 별칭은 별도의 예외 형이었습니다.

C name	Python name	노트
PyObject ***PyExc_EnvironmentError** *Part of the* 안정 ABI.	`OSError`	
PyObject ***PyExc_IOError** *Part of the* 안정 ABI.	`OSError`	
PyObject ***PyExc_WindowsError** *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로.	`OSError`	[win]

노트:

5.10.3 Warning types

C name	Python name
PyObject ***PyExc_Warning** *Part of the* 안정 ABI.	`Warning`
PyObject ***PyExc_BytesWarning** *Part of the* 안정 ABI.	`BytesWarning`
PyObject ***PyExc_DeprecationWarning** *Part of the* 안정 ABI.	`DeprecationWarning`
PyObject ***PyExc_EncodingWarning** *Part of the* 안정 ABI 버전 *3.10* 이후로.	`EncodingWarning`
PyObject ***PyExc_FutureWarning** *Part of the* 안정 ABI.	`FutureWarning`
PyObject ***PyExc_ImportWarning** *Part of the* 안정 ABI.	`ImportWarning`
PyObject ***PyExc_PendingDeprecationWarning** *Part of the* 안정 ABI.	`PendingDeprecationWarning`
PyObject ***PyExc_ResourceWarning** *Part of the* 안정 ABI 버전 *3.7* 이후로.	`ResourceWarning`
PyObject ***PyExc_RuntimeWarning** *Part of the* 안정 ABI.	`RuntimeWarning`
PyObject ***PyExc_SyntaxWarning** *Part of the* 안정 ABI.	`SyntaxWarning`
PyObject ***PyExc_UnicodeWarning** *Part of the* 안정 ABI.	`UnicodeWarning`
PyObject ***PyExc_UserWarning** *Part of the* 안정 ABI.	`UserWarning`

Added in version 3.2: *PyExc_ResourceWarning*.

Added in version 3.10: *PyExc_EncodingWarning*.

CHAPTER 6

Defining extension modules

A C extension for CPython is a shared library (for example, a `.so` file on Linux, `.pyd` DLL on Windows), which is loadable into the Python process (for example, it is compiled with compatible compiler settings), and which exports an *initialization function*.

To be importable by default (that is, by `importlib.machinery.ExtensionFileLoader`), the shared library must be available on `sys.path`, and must be named after the module name plus an extension listed in `importlib.machinery.EXTENSION_SUFFIXES`.

> **참고**
>
> Building, packaging and distributing extension modules is best done with third-party tools, and is out of scope of this document. One suitable tool is Setuptools, whose documentation can be found at https://setuptools.pypa.io/en/latest/setuptools.html.

Normally, the initialization function returns a module definition initialized using `PyModuleDef_Init()`. This allows splitting the creation process into several phases:

- Before any substantial code is executed, Python can determine which capabilities the module supports, and it can adjust the environment or refuse loading an incompatible extension.
- By default, Python itself creates the module object – that is, it does the equivalent of `object.__new__()` for classes. It also sets initial attributes like `__package__` and `__loader__`.
- Afterwards, the module object is initialized using extension-specific code – the equivalent of `__init__()` on classes.

This is called *multi-phase initialization* to distinguish it from the legacy (but still supported) *single-phase initialization* scheme, where the initialization function returns a fully constructed module. See the *single-phase-initialization section below* for details.

버전 3.5에서 변경: Added support for multi-phase initialization (**PEP 489**).

6.1 Multiple module instances

By default, extension modules are not singletons. For example, if the `sys.modules` entry is removed and the module is re-imported, a new module object is created, and typically populated with fresh method and type objects. The old module is subject to normal garbage collection. This mirrors the behavior of pure-Python modules.

Additional module instances may be created in *sub-interpreters* or after Python runtime reinitialization (`Py_Finalize()` and `Py_Initialize()`). In these cases, sharing Python objects between module instances would likely cause crashes or undefined behavior.

To avoid such issues, each instance of an extension module should be *isolated*: changes to one instance should not implicitly affect the others, and all state owned by the module, including references to Python objects, should be specific to a particular module instance. See isolating-extensions-howto for more details and a practical guide.

A simpler way to avoid these issues is raising an error on repeated initialization.

All modules are expected to support *sub-interpreters*, or otherwise explicitly signal a lack of support. This is usually achieved by isolation or blocking repeated initialization, as above. A module may also be limited to the main interpreter using the `Py_mod_multiple_interpreters` slot.

6.2 Initialization function

The initialization function defined by an extension module has the following signature:

PyObject ***PyInit_modulename**(void)

Its name should be `PyInit_<name>`, with `<name>` replaced by the name of the module.

For modules with ASCII-only names, the function must instead be named `PyInit_<name>`, with `<name>` replaced by the name of the module. When using *Multi-phase initialization*, non-ASCII module names are allowed. In this case, the initialization function name is `PyInitU_<name>`, with `<name>` encoded using Python's *punycode* encoding with hyphens replaced by underscores. In Python:

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-', b'_')
    return b'PyInit' + suffix
```

It is recommended to define the initialization function using a helper macro:

PyMODINIT_FUNC

Declare an extension module initialization function. This macro:

- specifies the `PyObject*` return type,
- adds any special linkage declarations required by the platform, and
- for C++, declares the function as `extern "C"`.

For example, a module called `spam` would be defined like this:

```
static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    ...
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

It is possible to export multiple modules from a single shared library by defining multiple initialization functions. However, importing them requires using symbolic links or a custom importer, because by default only the function corresponding to the filename is found. See the Multiple modules in one library section in **PEP 489** for details.

The initialization function is typically the only non-`static` item defined in the module's C source.

6.3 Multi-phase initialization

Normally, the *initialization function* (`PyInit_modulename`) returns a *`PyModuleDef`* instance with non-`NULL` *`m_slots`*. Before it is returned, the `PyModuleDef` instance must be initialized using the following function:

PyObject ***PyModuleDef_Init**(*PyModuleDef* *def)

Part of the 안정 ABI 버전 *3.5* 이후로. Ensure a module definition is a properly initialized Python object that correctly reports its type and a reference count.

Return *def* cast to `PyObject*`, or `NULL` if an error occurred.

Calling this function is required for *Multi-phase initialization*. It should not be used in other contexts.

Note that Python assumes that `PyModuleDef` structures are statically allocated. This function may return either a new reference or a borrowed one; this reference must not be released.

Added in version 3.5.

6.4 Legacy single-phase initialization

주의

Single-phase initialization is a legacy mechanism to initialize extension modules, with known drawbacks and design flaws. Extension module authors are encouraged to use multi-phase initialization instead.

In single-phase initialization, the *initialization function* (`PyInit_modulename`) should create, populate and return a module object. This is typically done using *`PyModule_Create()`* and functions like *`PyModule_AddObjectRef()`*.

Single-phase initialization differs from the *default* in the following ways:

- Single-phase modules are, or rather *contain*, “singletons”.

  When the module is first initialized, Python saves the contents of the module’s `__dict__` (that is, typically, the module’s functions and types).

  For subsequent imports, Python does not call the initialization function again. Instead, it creates a new module object with a new `__dict__`, and copies the saved contents to it. For example, given a single-phase module `_testsinglephase`[1] that defines a function `sum` and an exception class `error`:

  ```
  >>> import sys
  >>> import _testsinglephase as one
  >>> del sys.modules['_testsinglephase']
  >>> import _testsinglephase as two
  >>> one is two
  False
  >>> one.__dict__ is two.__dict__
  False
  >>> one.sum is two.sum
  True
  >>> one.error is two.error
  True
  ```

  The exact behavior should be considered a CPython implementation detail.

- To work around the fact that `PyInit_modulename` does not take a *spec* argument, some state of the import machinery is saved and applied to the first suitable module created during the `PyInit_modulename` call. Specifically, when a sub-module is imported, this mechanism prepends the parent package name to the name of the module.

[1] `_testsinglephase` is an internal module used in CPython’s self-test suite; your installation may or may not include it.

A single-phase `PyInit_modulename` function should create "its" module object as soon as possible, before any other module objects can be created.

- Non-ASCII module names (`PyInitU_modulename`) are not supported.
- Single-phase modules support module lookup functions like *PyState_FindModule()*.

CHAPTER 7

유틸리티

이 장의 함수들은 C 코드의 플랫폼 간 호환성 개선, C에서 파이썬 모듈 사용, 함수 인자의 구문 분석 및 C 값으로부터 파이썬 값을 구성하는 것에 이르기까지 다양한 유틸리티 작업을 수행합니다.

7.1 운영 체제 유틸리티

PyObject ***PyOS_FSPath**(*PyObject* *path)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.6* 이후로. *path*에 대한 파일 시스템 표현을 반환합니다. 객체가 `str`이나 `bytes` 객체이면, 새 강한 참조를 반환합니다. 객체가 `os.PathLike` 인터페이스를 구현하면, `__fspath__()`가 `str`나 `bytes` 객체일 때 반환됩니다. 그렇지 않으면, `TypeError`가 발생하고 `NULL`이 반환됩니다.

Added in version 3.6.

int **Py_FdIsInteractive**(FILE *fp, const char *filename)

이름이 *filename*인 표준 I/O 파일 *fp*를 대화식으로 간주하면 참(0이 아닙니다)을 반환합니다. `isatty(fileno(fp))`가 참인 파일의 경우입니다. *`PyConfig.interactive`*가 0이 아니면, 이 함수는 *filename* 포인터가 `NULL`이거나 이름이 문자열 `'<stdin>'`이나 `'???'` 중 하나와 같을 때도 참을 반환합니다.

This function must not be called before Python is initialized.

void **PyOS_BeforeFork**()

Part of the 안정 ABI *on platforms with fork*() 버전 *3.7* 이후로. 프로세스 포크 전에 내부 상태를 준비하는 함수. `fork()`나 현재 프로세스를 복제하는 유사한 함수를 호출하기 전에 호출해야 합니다. `fork()`가 정의된 시스템에서만 사용 가능합니다.

> **경고**
>
> C `fork()` 호출은 ("메인" 인터프리터의) "메인" 스레드에서만 이루어져야 합니다. `PyOS_BeforeFork()`도 마찬가지입니다.

Added in version 3.7.

void **PyOS_AfterFork_Parent**()

Part of the 안정 ABI *on platforms with fork*() 버전 *3.7* 이후로. 프로세스 포크 후 일부 내부 상태를 갱신하는 함수. 프로세스 복제가 성공했는지와 관계없이, `fork()`나 현재 프로세스를 복제하는 유사한

함수를 호출한 후 부모 프로세스에서 호출해야 합니다. `fork()` 가 정의된 시스템에서만 사용 가능합니다.

> **경고**
>
> C `fork()` 호출은 ("메인" 인터프리터의) "메인" 스레드에서만 이루어져야 합니다. `PyOS_AfterFork_Parent()` 도 마찬가지입니다.

Added in version 3.7.

void **PyOS_AfterFork_Child**()

Part of the 안정 ABI *on platforms with fork*() 버전 *3.7* 이후로. 프로세스 포크 후 내부 인터프리터 상태를 갱신하는 함수. `fork()` 나 현재 프로세스를 복제하는 유사한 함수를 호출한 후, 프로세스가 파이썬 인터프리터를 다시 호출할 가능성이 있으면 자식 프로세스에서 호출해야 합니다. `fork()` 가 정의된 시스템에서만 사용 가능합니다.

> **경고**
>
> C `fork()` 호출은 ("메인" 인터프리터의) "메인" 스레드에서만 이루어져야 합니다. `PyOS_AfterFork_Child()` 도 마찬가지입니다.

Added in version 3.7.

> **더 보기**
>
> `os.register_at_fork()` 를 사용하면 *PyOS_BeforeFork()*, *PyOS_AfterFork_Parent()* 및 *PyOS_AfterFork_Child()* 에서 호출될 사용자 정의 파이썬 함수를 등록할 수 있습니다.

void **PyOS_AfterFork**()

Part of the 안정 ABI *on platforms with fork*(). 프로세스 포크 후 일부 내부 상태를 갱신하는 함수; 파이썬 인터프리터가 계속 사용된다면 새로운 프로세스에서 호출되어야 합니다. 새 실행 파일이 새 프로세스에 로드되면, 이 함수를 호출할 필요가 없습니다.

버전 3.7부터 폐지됨: 이 함수는 *PyOS_AfterFork_Child()* 로 대체되었습니다.

int **PyOS_CheckStack**()

Part of the 안정 ABI *on platforms with USE_STACKCHECK* 버전 *3.7* 이후로. 인터프리터에 스택 공간이 부족하면 참을 반환합니다. 이것은 신뢰성 있는 확인이지만, `USE_STACKCHECK` 이 정의되어 있을 때만 사용할 수 있습니다 (현재 마이크로소프트 Visual C++ 컴파일러를 사용하는 특정 버전의 윈도우에서). `USE_STACKCHECK` 은 자동으로 정의됩니다; 여러분 자신의 코드에서 정의를 변경해서는 안 됩니다.

typedef void (***PyOS_sighandler_t**)(int)

Part of the 안정 ABI.

PyOS_sighandler_t **PyOS_getsig**(int i)

Part of the 안정 ABI. 시그널 *i* 에 대한 현재 시그널 처리기를 반환합니다. 이것은 `sigaction()` 이나 `signal()` 을 감싸는 얇은 래퍼입니다. 그 함수들을 직접 호출하지 마십시오!

PyOS_sighandler_t **PyOS_setsig**(int i, *PyOS_sighandler_t* h)

Part of the 안정 ABI. 시그널 *i* 의 시그널 처리기를 *h* 로 설정합니다; 이전 시그널 처리기를 반환합니다. 이것은 `sigaction()` 이나 `signal()` 을 감싸는 얇은 래퍼입니다. 그 함수들을 직접 호출하지 마십시오!

wchar_t ***Py_DecodeLocale**(const char *arg, size_t *size)

Part of the 안정 ABI 버전 *3.7* 이후로.

경고

This function should not be called directly: use the *PyConfig* API with the *PyConfig_SetBytesString()* function which ensures that *Python is preinitialized*.

This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured: see the *Py_PreInitialize()* function.

파일시스템 인코딩과 에러 처리기로 바이트열을 디코딩합니다. 에러 처리기가 surrogateescape 에러 처리기면, 디코딩할 수 없는 바이트열은 U+DC80..U+DCFF 범위의 문자로 디코딩됩니다; 그리고, 바이트 시퀀스를 서로게이트 문자로 디코딩 할 수 있으면, 이들을 디코딩하는 대신 surrogateescape 에러 처리기를 사용하여 바이트열를 이스케이프 합니다.

새로 할당된 와이드 문자(wide character) 문자열에 대한 포인터를 반환합니다, 메모리를 해제하려면 *PyMem_RawFree()*를 사용하십시오. size가 `NULL`이 아니면, 널 문자를 제외한 와이드 문자 수를 `*size`에 기록합니다.

디코딩 에러나 메모리 할당 에러 시 `NULL`을 반환합니다. *size*가 `NULL`이 아니면, 메모리 에러 시 `*size`가 `(size_t)-1`로 설정되고, 디코딩 에러 시 `(size_t)-2`로 설정됩니다.

The *filesystem encoding and error handler* are selected by *PyConfig_Read()*: see *filesystem_encoding* and *filesystem_errors* members of *PyConfig*.

C 라이브러리에 버그가 없으면, 디코딩 에러가 발생하지 않아야 합니다.

문자열을 바이트열로 다시 인코딩하려면 *Py_EncodeLocale()* 함수를 사용하십시오.

더 보기

*PyUnicode_DecodeFSDefaultAndSize()*와 *PyUnicode_DecodeLocaleAndSize()* 함수.

Added in version 3.5.

버전 3.7에서 변경: 이 함수는 이제 파이썬 UTF-8 모드에서 UTF-8 인코딩을 사용합니다.

버전 3.8에서 변경: 이 함수는 이제 윈도우에서 *PyPreConfig.legacy_windows_fs_encoding*가 0이면 UTF-8 인코딩을 사용합니다;

char ***Py_EncodeLocale**(const wchar_t *text, size_t *error_pos)

Part of the 안정 ABI 버전 *3.7* 이후로. 와이드 문자(wide character) 문자열을 파일시스템 인코딩과 에러 처리기로 인코딩합니다. 에러 처리기가 surrogateescape 에러 처리기면 U+DC80..U+DCFF 범위의 서로게이트 문자는 바이트 0x80..0xFF로 변환됩니다.

새로 할당된 바이트열에 대한 포인터를 반환합니다, 메모리를 해제하려면 *PyMem_Free()*를 사용하십시오. 인코딩 에러나 메모리 할당 에러 시 `NULL`을 반환합니다.

error_pos가 `NULL`이 아니면, `*error_pos`는 성공 시 `(size_t)-1`로 설정되고, 인코딩 에러 시 유효하지 않은 문자의 인덱스로 설정됩니다.

The *filesystem encoding and error handler* are selected by *PyConfig_Read()*: see *filesystem_encoding* and *filesystem_errors* members of *PyConfig*.

바이트열을 와이드 문자 문자열로 다시 디코딩하려면 *Py_DecodeLocale()* 함수를 사용하십시오.

경고

This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured: see the *Py_PreInitialize()* function.

> **더 보기**
>
> *PyUnicode_EncodeFSDefault()*와 *PyUnicode_EncodeLocale()* 함수.

Added in version 3.5.

버전 3.7에서 변경: 이 함수는 이제 파이썬 UTF-8 모드에서 UTF-8 인코딩을 사용합니다.

버전 3.8에서 변경: 이 함수는 이제 윈도우에서 *PyPreConfig.legacy_windows_fs_encoding*이 0이면 UTF-8 인코딩을 사용합니다.

FILE ***Py_fopen**(*PyObject* *path, const char *mode)

Similar to `fopen()`, but *path* is a Python object and an exception is set on error.

path must be a `str` object, a `bytes` object, or a *path-like object*.

On success, return the new file pointer. On error, set an exception and return `NULL`.

The file must be closed by *Py_fclose()* rather than calling directly `fclose()`.

The file descriptor is created non-inheritable (**PEP 446**).

The caller must have an *attached thread state*.

Added in version 3.14.

int **Py_fclose**(FILE *file)

Close a file that was opened by *Py_fopen()*.

On success, return `0`. On error, return `EOF` and `errno` is set to indicate the error. In either case, any further access (including another call to *Py_fclose()*) to the stream results in undefined behavior.

Added in version 3.14.

7.2 시스템 함수

`sys` 모듈의 기능을 C 코드에서 액세스 할 수 있게 하는 유틸리티 함수입니다. 모두 내부 스레드 상태 구조체에 포함된 현재 인터프리터 스레드의 `sys` 모듈의 딕셔너리에 작동합니다.

PyObject ***PySys_GetObject**(const char *name)

반환값: 빌린 참조. *Part of the* 안정 ABI. `sys` 모듈에서 객체 *name*을 반환하거나, 존재하지 않으면 예외를 설정하지 않고 `NULL`을 반환합니다.

int **PySys_SetObject**(const char *name, *PyObject* *v)

Part of the 안정 ABI. *v*가 *NULL*이 아닌 한 `sys` 모듈의 *name*을 *v*로 설정합니다. `NULL`이면 *name*은 sys 모듈에서 삭제됩니다. 성공하면 `0`, 에러 시 `-1`을 반환합니다.

void **PySys_ResetWarnOptions**()

Part of the 안정 ABI. `sys.warnoptions`를 빈 리스트로 재설정합니다. 이 함수는 *Py_Initialize()* 이전에 호출할 수 있습니다.

Deprecated since version 3.13, will be removed in version 3.15: Clear `sys.warnoptions` and `warnings.filters` instead.

void **PySys_WriteStdout**(const char *format, ...)

Part of the 안정 ABI. *format*으로 기술되는 출력 문자열을 `sys.stdout`에 기록합니다. 잘림이 발생하더라도 예외는 발생하지 않습니다 (아래를 참조하십시오).

*format*은 포맷된 출력 문자열의 총 크기를 1000바이트 이하로 제한해야 합니다 – 1000바이트 이후에는, 출력 문자열이 잘립니다. 특히, 이것은 무제한 "%s" 포맷이 있어서는 안 됨을 의미합니다; "%.<N>s"를 사용하여 제한해야 합니다, 여기서 <N>은 <N>에 다른 포맷된 텍스트의 최대 크기를 더할 때 1000바이트를 초과하지 않도록 계산된 십진수입니다. 또한 "%f"도 주의하십시오, 아주 큰 숫자는 수백 자리를 인쇄할 수 있습니다.

문제가 발생하거나, `sys.stdout`가 설정되어 있지 않으면, 포맷된 메시지는 실제(C 수준) *stdout*에 기록됩니다.

void **PySys_WriteStderr**(const char *format, ...)

Part of the 안정 ABI. *PySys_WriteStdout()*과 같지만, 대신 `sys.stderr`이나 *stderr*에 씁니다.

void **PySys_FormatStdout**(const char *format, ...)

Part of the 안정 ABI. PySys_WriteStdout()과 유사한 함수이지만, 메시지를 *PyUnicode_FromFormatV()*를 사용하여 포맷하고 메시지를 임의의 길이로 자르지 않습니다.

Added in version 3.2.

void **PySys_FormatStderr**(const char *format, ...)

Part of the 안정 ABI. *PySys_FormatStdout()*과 같지만, 대신 `sys.stderr`이나 *stderr*에 씁니다.

Added in version 3.2.

PyObject ***PySys_GetXOptions**()

반환값: 빌린 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. `sys._xoptions`와 유사하게, `-X` 옵션의 현재 딕셔너리를 반환합니다. 에러가 발생하면, `NULL`이 반환되고 예외가 설정됩니다.

Added in version 3.2.

int **PySys_Audit**(const char *event, const char *format, ...)

Part of the 안정 ABI 버전 *3.13* 이후로. 모든 활성 훅으로 감사 이벤트를 발생시킵니다. 성공 시 0을 반환하고 실패 시 예외를 설정하여 0이 아닌 값을 반환합니다.

The *event* string argument must not be *NULL*.

훅이 추가되었으면, *format*과 기타 인자를 사용하여 전달할 튜플을 구성합니다. `N` 외에도, *Py_BuildValue()*에서 사용된 것과 같은 포맷 문자를 사용할 수 있습니다. 빌드된 값이 튜플이 아니면, 단일 요소 튜플에 추가됩니다.

`N` 포맷 옵션은 사용하지 않아야 합니다. 참조를 소비하지만, 이 함수에 대한 인자가 소비될지를 알 방법이 없기 때문에, 사용하면 참조 누수가 발생할 수 있습니다.

`PY_SSIZE_T_CLEAN`이 정의되었는지와 관계없이, `#` 포맷 문자는 항상 *Py_ssize_t*로 처리되어야 합니다.

`sys.audit()`은 파이썬 코드와 동일한 기능을 수행합니다.

See also *PySys_AuditTuple()*.

Added in version 3.8.

버전 3.8.2에서 변경: `#` 포맷 문자에 대해 *Py_ssize_t*를 요구합니다. 이전에는, 피할 수 없는 폐지 경고가 발생했습니다.

int **PySys_AuditTuple**(const char *event, *PyObject* *args)

Part of the 안정 ABI 버전 *3.13* 이후로. Similar to *PySys_Audit()*, but pass arguments as a Python object. *args* must be a `tuple`. To pass no arguments, *args* can be *NULL*.

Added in version 3.13.

int **PySys_AddAuditHook**(*Py_AuditHookFunction* hook, void *userData)

활성 감사 훅 리스트에 콜러블 *hook*을 추가합니다. 성공하면 0을, 실패하면 0이 아닌 값을 반환합니다. 런타임이 초기화되었으면, 실패 시 에러도 설정합니다. 이 API를 통해 추가된 훅은 런타임이 만든 모든 인터프리터에 대해 호출됩니다.

userData 포인터는 훅 함수로 전달됩니다. 훅 함수는 다른 런타임에서 호출될 수 있어서, 이 포인터는 파이썬 상태를 직접 참조하면 안 됩니다.

이 함수는 *Py_Initialize()* 이전에 호출해도 안전합니다. 런타임 초기화 후 호출되면, 기존 감사 훅에 알리고 `Exception`에서 서브 클래싱 된 에러를 발생 시켜 조용히 연산을 중단할 수 있습니다 (다른 에러는 억제되지(silenced) 않습니다).

The hook function is always called with an *attached thread state* by the Python interpreter that raised the event.

감사에 대한 자세한 설명은 **PEP 578**을 참조하십시오. 이벤트를 발생시키는 런타임과 표준 라이브러리의 함수는 감사 이벤트 표에 나열되어 있습니다. 자세한 내용은 각 함수 설명서에 있습니다.

인터프리터가 초기화되면, 이 함수는 인자 없이 감사 이벤트 `sys.addaudithook`을 발생시킵니다. 기존 훅이 `Exception`에서 파생된 예외를 발생시키면, 새 훅이 추가되지 않고 예외가 지워집니다. 결과적으로, 호출자가 기존의 모든 훅을 제어하지 않는 한 훅이 추가되었다고 가정할 수 없습니다.

typedef int (***Py_AuditHookFunction**)(const char *event, *PyObject* *args, void *userData)

The type of the hook function. *event* is the C string event argument passed to *PySys_Audit()* or *PySys_AuditTuple()*. *args* is guaranteed to be a *PyTupleObject*. *userData* is the argument passed to PySys_AddAuditHook().

Added in version 3.8.

7.3 프로세스 제어

void **Py_FatalError**(const char *message)

Part of the 안정 ABI. 치명적인 에러 메시지를 인쇄하고 프로세스를 죽입니다. 아무런 정리도 수행되지 않습니다. 이 함수는 파이썬 인터프리터를 계속 사용하는 것이 위험한 조건이 감지되었을 때만 호출해야 합니다; 예를 들어, 객체 관리가 손상된 것으로 보일 때. 유닉스에서는, 표준 C 라이브러리 함수 `abort()`가 호출되어 `core` 파일을 생성하려고 시도합니다.

`Py_LIMITED_API` 매크로가 정의되어 있지 않은 한, `Py_FatalError()` 함수는 현재 함수의 이름을 자동으로 로그 하는 매크로로 대체됩니다.

버전 3.9에서 변경: 함수 이름을 자동으로 로그 합니다.

void **Py_Exit**(int status)

Part of the 안정 ABI. 현재 프로세스를 종료합니다. 이것은 *Py_FinalizeEx()*를 호출한 다음 표준 C 라이브러리 함수 `exit(status)`를 호출합니다. *Py_FinalizeEx()*가 에러를 표시하면, 종료 상태는 120으로 설정됩니다.

버전 3.6에서 변경: 파이널리제이션에서의 에러가 더는 무시되지 않습니다.

int **Py_AtExit**(void (*func)())

Part of the 안정 ABI. *Py_FinalizeEx()*가 호출할 정리 함수를 등록합니다. 정리 함수는 인자 없이 호출되며 값을 반환하지 않아야 합니다. 최대 32개의 정리 함수를 등록할 수 있습니다. 등록이 성공하면, *Py_AtExit()*는 `0`을 반환합니다; 실패하면 `-1`을 반환합니다. 마지막에 등록된 정리 함수가 먼저 호출됩니다. 각 정리 함수는 최대 한 번 호출됩니다. 정리 함수 전에 파이썬의 내부 파이널리제이션이 완료되기 때문에, *func*에서 파이썬 API를 호출하면 안 됩니다.

더 보기

PyUnstable_AtExit() for passing a `void *data` argument.

7.4 모듈 임포트 하기

PyObject ***PyImport_ImportModule**(const char *name)

반환값: 새 참조. *Part of the* 안정 ABI. This is a wrapper around *PyImport_Import()* which takes a `const char*` as an argument instead of a *PyObject**.

PyObject ***PyImport_ImportModuleNoBlock**(const char *name)

반환값: 새 참조. *Part of the* 안정 ABI. 이 함수는 *PyImport_ImportModule()*의 폐지된 별칭입니다.

버전 3.3에서 변경: 이 기능은 다른 스레드가 임포트 잠금을 보유한 경우 즉시 실패했었습니다. 그러나 파이썬 3.3에서는, 잠금 방식이 대부분의 목적에서 모듈 단위 잠금으로 전환되었기 때문에, 이 함수의 특수한 동작은 더는 필요하지 않습니다.

Deprecated since version 3.13, will be removed in version 3.15: *PyImport_ImportModule()*을 대신 사용하십시오.

PyObject ***PyImport_ImportModuleEx**(const char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist)

반환값: 새 참조. 모듈을 임포트 합니다. 내장 파이썬 함수 `__import__()`를 통해 가장 잘 설명할 수 있습니다.

반환 값은 임포트 된 모듈이나 최상위 패키지에 대한 새로운 참조, 또는 실패 시 예외가 설정된 `NULL`입니다. `__import__()`와 마찬가지로, 비어 있지 않은 *fromlist*가 제공되지 않는 한, 패키지의 서브모듈이 요청되었을 때의 반환 값은 최상위 패키지입니다.

임포트 실패는 *PyImport_ImportModule()*처럼 불완전한 모듈 객체를 제거합니다.

PyObject ***PyImport_ImportModuleLevelObject**(*PyObject* *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. 모듈을 임포트 합니다. 표준 `__import__()` 함수가 이 함수를 직접 호출하기 때문에, 내장 파이썬 함수 `__import__()`를 통해 가장 잘 설명할 수 있습니다.

반환 값은 임포트 된 모듈이나 최상위 패키지에 대한 새로운 참조, 또는 실패 시 예외가 설정된 `NULL`입니다. `__import__()`와 마찬가지로, 비어 있지 않은 *fromlist*가 제공되지 않는 한, 패키지의 서브모듈이 요청되었을 때의 반환 값은 최상위 패키지입니다.

Added in version 3.3.

PyObject ***PyImport_ImportModuleLevel**(const char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)

반환값: 새 참조. *Part of the* 안정 ABI. *PyImport_ImportModuleLevelObject()*와 비슷하지만, name은 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

버전 3.3에서 변경: *level*의 음수 값은 더는 허용되지 않습니다.

PyObject ***PyImport_Import**(*PyObject* *name)

반환값: 새 참조. *Part of the* 안정 ABI. 이것은 현재 "임포트 훅 함수"를 호출하는 고수준 인터페이스입니다 (명시적인 *level* 0을 사용하는데, 절대 임포트를 뜻합니다). 현재 전역의 `__builtins__`에 있는 `__import__()` 함수를 호출합니다. 이는 현재 환경에 설치된 임포트 훅을 사용하여 임포트가 수행됨을 의미합니다.

이 함수는 항상 절대 임포트를 사용합니다.

PyObject ***PyImport_ReloadModule**(*PyObject* *m)

반환값: 새 참조. *Part of the* 안정 ABI. 모듈을 다시 로드(reload)합니다. 다시 로드된 모듈에 대한 참조를 반환하거나, 실패 시 예외가 설정된 `NULL`을 반환합니다 (이때 모듈은 여전히 존재합니다).

PyObject ***PyImport_AddModuleRef**(const char *name)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.13* 이후로. 모듈 이름에 해당하는 모듈 객체를 반환합니다.

name 인자는 `package.module` 형식일 수 있습니다. 먼저 모듈 딕셔너리에 있는지 확인하고, 없으면 새로 만들어 모듈 딕셔너리에 삽입합니다.

성공하면 모듈에 대한 강한 참조를 반환합니다. 실패 시 예외를 설정하고 `NULL`을 반환합니다.

The module name *name* is decoded from UTF-8.

이 함수는 모듈을 로드하거나 임포트 하지 않습니다; 모듈이 아직 로드되지 않았으면, 빈 모듈 객체를 얻게 됩니다. 모듈을 임포트 하려면 *PyImport_ImportModule()*이나 그 변형 중 하나를 사용하십시오. *name*에서 점으로 구분된 이름으로 암시된 패키지 구조는 이미 존재하지 않는다면 만들어지지 않습니다.

Added in version 3.13.

PyObject ***PyImport_AddModuleObject**(*PyObject* *name)

반환값: 빌린 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. *PyImport_AddModuleRef()*와 비슷하지만, 빌린 참조를 반환하고 *name*은 `str` 객체입니다.

Added in version 3.3.

PyObject ***PyImport_AddModule**(const char *name)

반환값: 빌린 참조. *Part of the* 안정 ABI. Similar to `PyImport_AddModuleRef()`, but return a *borrowed reference*.

PyObject ***PyImport_ExecCodeModule**(const char *name, *PyObject* *co)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 모듈 이름(name)(`package.module` 형식일 수 있습니다)과 파이썬 바이트 코드 파일에서 읽거나 내장 함수 `compile()`로 얻은 코드 객체로, 모듈을 로드합니다. 모듈 객체에 대한 새로운 참조를 반환하거나, 또는 에러가 발생하면 예외가 설정된 `NULL`을 반환합니다. 에러가 발생하면 `sys.modules`에서 *name*이 제거됩니다, `PyImport_ExecCodeModule()`에 진입할 때 *name*이 `sys.modules`에 이미 있어도 그렇습니다. `sys.modules`에 불완전하게 초기화된 모듈을 남겨 두는 것은 위험합니다, 그러한 모듈을 임포트 할 때 모듈 객체가 알 수 없는 (그리고 아마도 모듈 작성자의 의도에 비추어볼 때 손상된) 상태에 있음을 알 방법이 없기 때문입니다.

모듈의 `__spec__`과 `__loader__`는 아직 설정되지 않았다면 적절한 값으로 설정됩니다. 스펙의 로더는 모듈의 `__loader__`(설정되었다면)로 설정되고, 그렇지 않으면 `SourceFileLoader`의 인스턴스로 설정됩니다.

모듈의 `__file__` 어트리뷰트는 코드 객체의 `co_filename`으로 설정됩니다. 해당한다면, `__cached__`도 설정됩니다.

이 함수는 이미 임포트 되었다면 모듈을 다시 로드합니다. 모듈을 다시 로드하는 의도된 방법은 `PyImport_ReloadModule()`을 참조하십시오.

*name*이 `package.module` 형식의 점으로 구분된 이름을 가리키면, 이미 만들어지지 않은 패키지 구조는 여전히 만들어지지 않습니다.

`PyImport_ExecCodeModuleEx()`와 `PyImport_ExecCodeModuleWithPathnames()`도 참조하십시오.

버전 3.12에서 변경: The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

PyObject ***PyImport_ExecCodeModuleEx**(const char *name, *PyObject* *co, const char *pathname)

반환값: 새 참조. *Part of the* 안정 ABI. `PyImport_ExecCodeModule()`과 유사하지만, 모듈 객체의 `__file__` 어트리뷰트는 `NULL`이 아니라면 *pathname*으로 설정됩니다.

`PyImport_ExecCodeModuleWithPathnames()`도 참조하십시오.

PyObject ***PyImport_ExecCodeModuleObject**(*PyObject* *name, *PyObject* *co, *PyObject* *pathname, *PyObject* *cpathname)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. `PyImport_ExecCodeModuleEx()`와 유사하지만, 모듈 객체의 `__cached__` 어트리뷰트는 `NULL`이 아니라면 *cpathname*으로 설정됩니다. 세 가지 함수 중 이것이 선호되는 것입니다.

Added in version 3.3.

버전 3.12에서 변경: Setting `__cached__` is deprecated. See `ModuleSpec` for alternatives.

PyObject ***PyImport_ExecCodeModuleWithPathnames**(const char *name, *PyObject* *co, const char *pathname, const char *cpathname)

반환값: 새 참조. *Part of the* 안정 ABI. `PyImport_ExecCodeModuleObject()`와 유사하지만, *name*, *pathname* 및 *cpathname*은 UTF-8로 인코딩된 문자열입니다. *pathname*의 값이 `NULL`로 설정된 경우 어떤 값이 *cpathname*에서 와야하는지 알아내려고 합니다.

Added in version 3.2.

버전 3.3에서 변경: 바이트 코드 경로만 제공되면 소스 경로를 계산할 때 `imp.source_from_cache()`를 사용합니다.

버전 3.12에서 변경: No longer uses the removed `imp` module.

long **PyImport_GetMagicNumber**()

Part of the 안정 ABI. 파이썬 바이트 코드 파일(일명 `.pyc` 파일)의 매직 번호(magic number)를 반환합니다. 매직 번호는 바이트 코드 파일의 처음 4바이트에 리틀 엔디안 바이트 순서로 존재해야 합니다. 에러 시 `-1`을 반환합니다.

버전 3.3에서 변경: 실패 시 -1을 반환합니다.

const char *PyImport_GetMagicTag()

Part of the 안정 ABI. **PEP 3147** 형식 파이썬 바이트 코드 파일 이름의 매직 태그 문자열을 반환합니다. sys.implementation.cache_tag의 값은 신뢰할 수 있고 이 함수 대신 사용해야 함에 유의하십시오.

Added in version 3.2.

PyObject *PyImport_GetModuleDict()

반환값: 빌린 참조. *Part of the* 안정 ABI. 모듈 관리에 사용되는 딕셔너리(일명 sys.modules)를 반환합니다. 이것은 인터프리터마다 존재하는 변수임에 유의하십시오.

PyObject *PyImport_GetModule(*PyObject* *name)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.8* 이후로. 주어진 이름으로 이미 임포트 된 모듈을 반환합니다. 모듈이 아직 임포트 되지 않았다면 NULL을 반환하지만 에러는 설정하지 않습니다. 조회에 실패하면 NULL을 반환하고 에러를 설정합니다.

Added in version 3.7.

PyObject *PyImport_GetImporter(*PyObject* *path)

반환값: 새 참조. *Part of the* 안정 ABI. sys.path/pkg.__path__ 항목 *path*를 위한 파인더 객체를 반환합니다, sys.path_importer_cache 딕셔너리에서 꺼낼 수도 있습니다. 아직 캐시 되지 않았으면, 경로 항목을 처리할 수 있는 훅이 발견될 때까지 sys.path_hooks를 탐색합니다. 훅이 없으면 None을 반환합니다; 이것은 호출자에게 경로 기반 파인더가 이 경로 항목에 대한 파인더를 찾을 수 없음을 알려줍니다. sys.path_importer_cache에 결과를 캐시 합니다. 파인더 객체에 대한 새로운 참조를 반환합니다.

int PyImport_ImportFrozenModuleObject(*PyObject* *name)

Part of the 안정 ABI 버전 *3.7* 이후로. *name*이라는 이름의 프로즌 모듈(frozen module)을 로드합니다. 성공하면 1을, 모듈을 찾지 못하면 0을, 초기화에 실패하면 예외를 설정하고 -1을 반환합니다. 로드가 성공할 때 임포트 된 모듈에 액세스하려면 *PyImport_ImportModule()*을 사용하십시오. (잘못된 이름에 주의하십시오 — 이 함수는 모듈이 이미 임포트 되었을 때 다시 로드합니다.)

Added in version 3.3.

버전 3.4에서 변경: __file__ 어트리뷰트는 더는 모듈에 설정되지 않습니다.

int PyImport_ImportFrozenModule(const char *name)

Part of the 안정 ABI. *PyImport_ImportFrozenModuleObject()*와 비슷하지만, name은 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

struct _frozen

이것은 **freeze** 유틸리티(파이썬 소스 배포의 Tools/freeze/를 참조하십시오)가 생성한 프로즌 모듈 디스크립터를 위한 구조체 형 정의입니다. Include/import.h에 있는 정의는 다음과 같습니다:

```
struct _frozen {
    const char *name;
    const unsigned char *code;
    int size;
    bool is_package;
};
```

버전 3.11에서 변경: The new is_package field indicates whether the module is a package or not. This replaces setting the size field to a negative value.

const struct *_frozen* *PyImport_FrozenModules

이 포인터는 *_frozen* 레코드의 배열을 가리키도록 초기화되는데, 멤버가 모두 NULL이나 0인 레코드로 끝납니다. 프로즌 모듈이 임포트 될 때, 이 테이블에서 검색됩니다. 제삼자 코드는 이것을 사용하여 동적으로 생성된 프로즌 모듈 컬렉션을 제공할 수 있습니다.

int PyImport_AppendInittab(const char *name, *PyObject* *(*initfunc)(void))

Part of the 안정 ABI. 기존의 내장 모듈 테이블에 단일 모듈을 추가합니다. 이것은 *PyImport_ExtendInittab()*을 감싸는 편리한 래퍼인데, 테이블을 확장할 수 없으면 -1을 반환

합니다. 새 모듈은 *name*이라는 이름으로 임포트 될 수 있으며, *initfunc* 함수를 처음 시도한 임포트에서 호출되는 초기화 함수로 사용합니다. *Py_Initialize()* 전에 호출해야 합니다.

struct **_inittab**

내장 모듈 목록에 있는 단일 항목을 기술하는 구조체. 파이썬을 내장하는 프로그램은 *PyImport_ExtendInittab()*과 함께 이러한 구조체의 배열을 사용하여 추가 내장 모듈을 제공 할 수 있습니다. 구조체는 두 개의 멤버로 구성됩니다:

const char ***name**

모듈 이름, ASCII로 인코딩된 문자열.

PyObject *(***initfunc**)(void)

인터프리터에 내장된 모듈을 위한 초기화 함수.

int **PyImport_ExtendInittab**(struct *_inittab* *newtab)

내장 모듈 테이블에 모듈 컬렉션을 추가합니다. *newtab* 배열은 *name* 필드에 `NULL`을 포함하는 센티넬 (sentinel) 항목으로 끝나야 합니다; 센티넬 값을 제공하지 않으면 메모리 오류가 발생할 수 있습니다. 성공하면 `0`을, 내부 테이블을 확장하기 위한 메모리가 충분하지 않으면 `-1`을 반환합니다. 실패하면, 내부 테이블에 모듈이 추가되지 않습니다. *Py_Initialize()* 전에 호출해야 합니다.

If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

PyObject ***PyImport_ImportModuleAttr**(*PyObject* *mod_name, *PyObject* *attr_name)

반환값: 새 참조. Import the module *mod_name* and get its attribute *attr_name*.

Names must be Python `str` objects.

Helper function combining *PyImport_Import()* and *PyObject_GetAttr()*. For example, it can raise `ImportError` if the module is not found, and `AttributeError` if the attribute doesn' t exist.

Added in version 3.14.

PyObject ***PyImport_ImportModuleAttrString**(const char *mod_name, const char *attr_name)

반환값: 새 참조. Similar to *PyImport_ImportModuleAttr()*, but names are UTF-8 encoded strings instead of Python `str` objects.

Added in version 3.14.

7.5 데이터 마샬링 지원

이러한 루틴은 C 코드가 `marshal` 모듈과 같은 데이터 형식을 사용하여 직렬화된 객체로 작업 할 수 있도록 합니다. 직렬화 형식으로 데이터를 쓰는 함수와 데이터를 다시 읽는 데 사용할 수 있는 추가 함수가 있습니다. 마샬링 된 데이터를 저장하는 데 사용되는 파일은 바이너리 모드로 열어야 합니다.

숫자 값은 최하위 바이트가 먼저 저장됩니다.

The module supports several versions of the data format; see the `Python module documentation` for details.

Py_MARSHAL_VERSION

The current format version. See `marshal.version`.

void **PyMarshal_WriteLongToFile**(long value, FILE *file, int version)

`long` 정수 *value*를 *file*로 마샬합니다. *value*의 최하위 32비트 만 기록합니다; 기본 `long` 형의 크기와 관계없이. *version*은 파일 형식을 나타냅니다.

This function can fail, in which case it sets the error indicator. Use *PyErr_Occurred()* to check for that.

void **PyMarshal_WriteObjectToFile**(*PyObject* *value, FILE *file, int version)

파이썬 객체 *value*를 *file*로 마샬합니다. *version*은 파일 형식을 나타냅니다.

This function can fail, in which case it sets the error indicator. Use *PyErr_Occurred()* to check for that.

PyObject ***PyMarshal_WriteObjectToString**(*PyObject* *value, int version)

반환값: 새 참조. 마샬된 *value* 표현을 포함한 바이트열 객체를 반환합니다. *version*은 파일 형식을 나타냅니다.

다음 함수를 사용하면 마샬된 값을 다시 읽을 수 있습니다.

long **PyMarshal_ReadLongFromFile**(FILE *file)

읽기 위해 열린 `FILE*`의 데이터 스트림에서 C `long`을 반환합니다. 이 함수를 사용하면 `long`의 기본 크기와 관계없이 32비트 값만 읽을 수 있습니다.

에러 시, 적절한 예외(`EOFError`)를 설정하고 `-1`을 반환합니다.

int **PyMarshal_ReadShortFromFile**(FILE *file)

읽기 위해 열린 `FILE*`의 데이터 스트림에서 C `short`를 반환합니다. 이 함수를 사용하면 `short`의 기본 크기와 관계없이 16비트 값만 읽을 수 있습니다.

에러 시, 적절한 예외(`EOFError`)를 설정하고 `-1`을 반환합니다.

PyObject ***PyMarshal_ReadObjectFromFile**(FILE *file)

반환값: 새 참조. 읽기 위해 열린 `FILE*`의 데이터 스트림에서 파이썬 객체를 반환합니다.

에러 시, 적절한 예외(`EOFError`, `ValueError` 또는 `TypeError`)를 설정하고 `NULL`을 반환합니다.

PyObject ***PyMarshal_ReadLastObjectFromFile**(FILE *file)

반환값: 새 참조. 읽기 위해 열린 `FILE*`의 데이터 스트림에서 파이썬 객체를 반환합니다. *PyMarshal_ReadObjectFromFile()*와 달리, 이 함수는 더는 파일에서 객체를 읽지 않을 것이라고 가정함으로써, 파일 데이터를 메모리에 적극적으로 로드 할 수 있고, 파일에서 한 바이트씩 읽는 대신 메모리에 있는 데이터에서 역 직렬화가 작동할 수 있습니다. 파일에서 어떤 것도 읽지 않을 것이라는 확신이 들 경우에만 이 변형을 사용하십시오.

에러 시, 적절한 예외(`EOFError`, `ValueError` 또는 `TypeError`)를 설정하고 `NULL`을 반환합니다.

PyObject ***PyMarshal_ReadObjectFromString**(const char *data, *Py_ssize_t* len)

반환값: 새 참조. *data*가 가리키는 *len* 바이트를 포함하는 바이트 버퍼의 데이터 스트림에서 파이썬 객체를 반환합니다.

에러 시, 적절한 예외(`EOFError`, `ValueError` 또는 `TypeError`)를 설정하고 `NULL`을 반환합니다.

7.6 인자 구문 분석과 값 구축

이 함수들은 자체 확장 함수와 메서드를 만들 때 유용합니다. 추가 정보와 예제는 extending-index에 있습니다.

설명된 이러한 함수 중 처음 세 개인 *PyArg_ParseTuple()*, *PyArg_ParseTupleAndKeywords()* 및 *PyArg_Parse()*는 모두 예상 인자에 관한 사항을 함수에 알리는 데 사용되는 포맷 문자열(*format strings*)을 사용합니다. 포맷 문자열은 이러한 각 함수에 대해 같은 문법을 사용합니다.

7.6.1 인자 구문 분석

포맷 문자열은 0개 이상의 "포맷 단위(format units)"로 구성됩니다. 포맷 단위는 하나의 파이썬 객체를 설명합니다; 일반적으로 단일 문자나 괄호로 묶인 포맷 단위 시퀀스입니다. 몇 가지 예외를 제외하고, 괄호로 묶인 시퀀스가 아닌 포맷 단위는 일반적으로 이러한 함수에 대한 단일 주소 인자에 대응합니다. 다음 설명에서, 인용된(quoted) 형식은 포맷 단위입니다; (둥근) 괄호 안의 항목은 포맷 단위와 일치하는 파이썬 객체 형입니다; [대괄호] 안의 항목은 주소를 전달해야 하는 C 변수의 형입니다.

문자열과 버퍼

참고

On Python 3.12 and older, the macro `PY_SSIZE_T_CLEAN` must be defined before including `Python.h` to use all `#` variants of formats (`s#`, `y#`, etc.) explained below. This is not necessary on Python 3.13 and later.

이러한 포맷을 사용하면 연속적인 메모리 청크로 객체에 액세스 할 수 있습니다. 반환된 유니코드나 바이트열 영역에 대한 원시 저장소를 제공할 필요가 없습니다.

달리 명시되지 않는 한, 버퍼는 NUL로 종료되지 않습니다.

There are three ways strings and buffers can be converted to C:

- `y*` 와 `s*` 같은 포맷은 *Py_buffer* 구조체를 채웁니다. 이는 하부 버퍼를 잠가서, 호출자가 *Py_BEGIN_ALLOW_THREADS* 블록 내에서도 가변 데이터의 크기가 조정되거나 파괴될 위험 없이 이후에 버퍼를 사용할 수 있습니다. 결과적으로, 데이터 처리를 마친 후 (또는 모든 조기 중단의 경우) *PyBuffer_Release()* 를 **호출해야 합니다**.
- The `es`, `es#`, `et` and `et#` formats allocate the result buffer. **You have to call** *PyMem_Free()* after you have finished processing the data (or in any early abort case).
- Other formats take a `str` or a read-only *bytes-like object*, such as `bytes`, and provide a `const char *` pointer to its buffer. In this case the buffer is "borrowed": it is managed by the corresponding Python object, and shares the lifetime of this object. You won' t have to release any memory yourself.

  To ensure that the underlying buffer may be safely borrowed, the object' s *PyBufferProcs.bf_releasebuffer* field must be `NULL`. This disallows common mutable objects such as `bytearray`, but also some read-only objects such as `memoryview` of `bytes`.

  Besides this `bf_releasebuffer` requirement, there is no check to verify whether the input object is immutable (e.g. whether it would honor a request for a writable buffer, or whether another thread can mutate the data).

s (str) [const char *]
유니코드 객체를 문자열에 대한 C 포인터로 변환합니다. 기존 문자열에 대한 포인터는 여러분이 주소를 전달한 문자 포인터 변수에 저장됩니다. C 문자열은 NUL로 종료됩니다. 파이썬 문자열은 내장된 널 코드 포인트를 포함하지 않아야 합니다; 그렇다면 `ValueError` 예외가 발생합니다. 유니코드 객체는 `'utf-8'` 인코딩을 사용하여 C 문자열로 변환됩니다. 이 변환이 실패하면, `UnicodeError`가 발생합니다.

> **참고**
>
> 이 포맷은 바이트열류 객체를 받아들이지 않습니다. 파일 시스템 경로를 받아들이고 이를 C 문자열로 변환하려면, *PyUnicode_FSConverter()* 를 *converter*로 `O&` 포맷을 사용하는 것이 좋습니다.

버전 3.5에서 변경: 이전에는, 파이썬 문자열에서 내장된 널 코드 포인트가 발견되면 `TypeError`가 발생했습니다.

s* (str 또는 바이트열류 객체) [Py_buffer]
이 포맷은 바이트열류 객체뿐만 아니라 유니코드 객체를 받아들입니다. 호출자가 제공한 *Py_buffer* 구조체를 채웁니다. 이 경우 결과 C 문자열은 내장된 NUL 바이트를 포함할 수 있습니다. 유니코드 객체는 `'utf-8'` 인코딩을 사용하여 C 문자열로 변환됩니다.

s# (str, 읽기 전용 바이트열류 객체) [const char *, *Py_ssize_t*]
빌린 버퍼를 제공한다는 점을 제외하면, `s*`와 같습니다. 결과는 두 개의 C 변수에 저장됩니다. 첫 번째 변수는 C 문자열에 대한 포인터이고, 두 번째 변수는 길이입니다. 문자열은 내장 널 바이트를 포함할 수 있습니다. 유니코드 객체는 `'utf-8'` 인코딩을 사용하여 C 문자열로 변환됩니다.

z (str 또는 None) [const char *]
`s`와 비슷하지만, 파이썬 객체가 `None`일 수도 있는데, 이 경우 C 포인터가 `NULL`로 설정됩니다.

z* (str, 바이트열류 객체 또는 None) [Py_buffer]
`s*`와 비슷하지만, 파이썬 객체는 `None`일 수도 있습니다, 이 경우 *Py_buffer* 구조체의 `buf` 멤버가 `NULL`로 설정됩니다.

z# (str, 읽기 전용 바이트열류 객체 또는 None) [const char *, *Py_ssize_t*]
`s#`와 비슷하지만, 파이썬 객체는 `None`일 수도 있습니다, 이 경우 C 포인터가 `NULL`로 설정됩니다.

y (읽기 전용 바이트열류 객체) [const char *]
이 포맷은 바이트열류 객체를 빌린 문자열에 대한 C 포인터로 변환합니다; 유니코드 객체를 받

아들이지 않습니다. 바이트열 버퍼는 내장 널 바이트를 포함하지 않아야 합니다; 만약 그렇다면, `ValueError` 예외가 발생합니다.

버전 3.5에서 변경: 이전에는, 바이트열 버퍼에서 내장 널 바이트가 발견되면 `TypeError`가 발생했습니다.

y* (바이트열류 객체) **[Py_buffer]**
`s*`의 이 변형은 유니코드 객체가 아니라 바이트열류 객체만 받아들입니다. **바이너리 데이터를 받아들이는 권장 방법입니다.**

y# (읽기 전용 바이트열류 객체) **[const char *, *Py_ssize_t*]**
`s#`의 이 변형은 유니코드 객체가 아니라 바이트열류 객체만 받아들입니다.

S (**bytes**) **[PyBytesObject *]**
변환을 시도하지 않고, 파이썬 객체가 `bytes` 객체일 것을 요구합니다. 객체가 바이트열 객체가 아니면 `TypeError`를 발생시킵니다. C 변수는 *PyObject**로 선언될 수도 있습니다.

Y (**bytearray**) **[PyByteArrayObject *]**
변환을 시도하지 않고, 파이썬 객체가 `bytearray` 객체일 것을 요구합니다. 객체가 `bytearray` 객체가 아니면 `TypeError`를 발생시킵니다. C 변수는 *PyObject**로 선언될 수도 있습니다.

U (**str**) **[PyObject *]**
변환을 시도하지 않고, 파이썬 객체가 유니코드 객체일 것을 요구합니다. 객체가 유니코드 객체가 아니면 `TypeError`를 발생시킵니다. C 변수는 *PyObject**로 선언될 수도 있습니다.

w* (읽기-쓰기 바이트열류 객체) **[Py_buffer]**
이 포맷은 읽기-쓰기 버퍼 인터페이스를 구현하는 모든 객체를 허용합니다. 호출자가 제공한 *Py_buffer* 구조체를 채웁니다. 버퍼에는 내장 널 바이트가 포함될 수 있습니다. 호출자는 버퍼로 할 일을 마치면 *PyBuffer_Release()*를 호출해야 합니다.

es (**str**) **[const char *encoding, char **buffer]**
`s`의 이 변형은 유니코드를 문자 버퍼로 인코딩하는 데 사용됩니다. 내장 NUL 바이트가 포함되지 않은 인코딩된 데이터에 대해서만 작동합니다.

이 포맷에는 두 개의 인자가 필요합니다. 첫 번째는 입력으로만 사용되며, 인코딩 이름을 가리키는 NUL 종료 문자열로 `const char*`이거나, `'utf-8'` 인코딩이 사용되도록 하는 `NULL`이어야 합니다. 명명된 인코딩이 파이썬에 알려지지 않았으면 예외가 발생합니다. 두 번째 인자는 `char**`여야 합니다; 참조하는 포인터의 값은 인자 텍스트의 내용이 있는 버퍼로 설정됩니다. 텍스트는 첫 번째 인자에 지정된 인코딩으로 인코딩됩니다.

*PyArg_ParseTuple()*은 필요한 크기의 버퍼를 할당하고, 인코딩된 데이터를 이 버퍼에 복사하고 새로 할당된 스토리지를 참조하도록 **buffer*를 조정합니다. 호출자는 사용 후에 할당된 버퍼를 해제하기 위해 *PyMem_Free()*를 호출해야 합니다.

et (**str**, **bytes** 또는 **bytearray**) **[const char *encoding, char **buffer]**
바이트 문자열 객체를 다시 코딩하지 않고 통과시킨다는 점을 제외하면 `es`와 같습니다. 대신, 구현은 바이트 문자열 객체가 매개 변수로 전달된 인코딩을 사용한다고 가정합니다.

es# (**str**) **[const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]**
`s#`의 이 변형은 유니코드를 문자 버퍼로 인코딩하는 데 사용됩니다. `es` 포맷과 달리, 이 변형은 NUL 문자를 포함하는 입력 데이터를 허용합니다.

세 가지 인자가 필요합니다. 첫 번째는 입력으로만 사용되며, 인코딩 이름을 가리키는 NUL 종료 문자열로 `const char*`이거나, `'utf-8'` 인코딩이 사용되도록 하는 `NULL`이어야 합니다. 명명된 인코딩이 파이썬에 알려지지 않았으면 예외가 발생합니다. 두 번째 인자는 `char**`여야 합니다; 참조하는 포인터의 값은 인자 텍스트의 내용이 있는 버퍼로 설정됩니다. 텍스트는 첫 번째 인자에 지정된 인코딩으로 인코딩됩니다. 세 번째 인자는 정수에 대한 포인터여야 합니다; 참조된 정수는 출력 버퍼의 바이트 수로 설정됩니다.

두 가지 작동 모드가 있습니다:

**buffer*가 `NULL` 포인터를 가리키면, 함수는 필요한 크기의 버퍼를 할당하고, 이 버퍼로 인코딩된 데이터를 복사하고 **buffer*를 새로 할당된 스토리지를 참조하도록 설정합니다. 호출자는 사용 후 할당된 버퍼를 해제하기 위해 *PyMem_Free()*를 호출해야 합니다.

*buffer*가 NULL이 아닌 포인터를 가리키면 (이미 할당된 버퍼), *PyArg_ParseTuple()*은 이 위치를 버퍼로 사용하고 **buffer_length*의 초깃값을 버퍼 크기로 해석합니다. 그런 다음 인코딩된 데이터를 버퍼에 복사하고 NUL 종료합니다. 버퍼가 충분히 크지 않으면, ValueError가 설정됩니다.

두 경우 모두, **buffer_length*는 후행 NUL 바이트를 제외한 인코딩된 데이터의 길이로 설정됩니다.

et# (str, bytes 또는 bytearray) [const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]
바이트 문자열 객체를 다시 코딩하지 않고 통과시킨다는 점을 제외하면 es#와 같습니다. 대신, 구현은 바이트 문자열 객체가 매개 변수로 전달된 인코딩을 사용한다고 가정합니다.

버전 3.12에서 변경: u, u#, Z, and Z# are removed because they used a legacy Py_UNICODE* representation.

숫자

These formats allow representing Python numbers or single characters as C numbers. Formats that require int, float or complex can also use the corresponding special methods __index__(), __float__() or __complex__() to convert the Python object to the required type.

For signed integer formats, OverflowError is raised if the value is out of range for the C type. For unsigned integer formats, no range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value.

b (int) [unsigned char]
음이 아닌 파이썬 정수를 부호 없는 작은 정수로 변환하고, C unsigned char에 저장합니다.

B (int) [unsigned char]
오버플로 검사 없이 파이썬 정수를 작은 정수로 변환하고, C unsigned char에 저장합니다.

h (int) [short int]
파이썬 정수를 C short int로 변환합니다.

H (int) [unsigned short int]
오버플로 검사 없이, 파이썬 정수를 C unsigned short int로 변환합니다.

i (int) [int]
파이썬 정수를 일반 C int로 변환합니다.

I (int) [unsigned int]
오버플로 검사 없이, 파이썬 정수를 C unsigned int로 변환합니다.

l (int) [long int]
파이썬 정수를 C long int로 변환합니다.

k (int) [unsigned long]
오버플로 검사 없이 파이썬 정수를 C unsigned long으로 변환합니다.

버전 3.14에서 변경: Use __index__() if available.

L (int) [long long]
파이썬 정수를 C long long으로 변환합니다.

K (int) [unsigned long long]
오버플로 검사 없이 파이썬 정수를 C unsigned long long으로 변환합니다.

버전 3.14에서 변경: Use __index__() if available.

n (int) [*Py_ssize_t*]
파이썬 정수를 C *Py_ssize_t*로 변환합니다.

c (길이 1의 bytes 또는 bytearray) [char]
길이가 1인 bytes나 bytearray 객체로 표시된, 파이썬 바이트를 C char로 변환합니다.

버전 3.3에서 변경: bytearray 객체를 허용합니다.

C (길이 1의 str) [int]
길이가 1인 str 객체로 표시된, 파이썬 문자를 C int로 변환합니다.

f (float) [float]
파이썬 부동 소수점 숫자를 C float로 변환합니다.

d (**float**) [**double**]
파이썬 부동 소수점 숫자를 C `double`로 변환합니다.

D (**complex**) [**Py_complex**]
파이썬 복소수를 C *Py_complex* 구조체로 변환합니다.

기타 객체

O (**object**) [**PyObject ***]
C 객체 포인터에 파이썬 객체를 (변환 없이) 저장합니다. 따라서 C 프로그램은 전달된 실제 객체를 받습니다. 객체에 대한 새로운 강한 참조는 생성되지 않습니다 (즉, 객체의 참조 횟수는 증가하지 않습니다). 저장된 포인터는 `NULL`이 아닙니다.

O! (**object**) [*typeobject*, **PyObject ***]
C 객체 포인터에 파이썬 객체를 저장합니다. 이것은 `O`와 유사하지만, 두 개의 C 인자를 취합니다: 첫 번째는 파이썬 형 객체의 주소이고, 두 번째는 객체 포인터가 저장되는 (*PyObject** 형의) C 변수의 주소입니다. 파이썬 객체가 필요한 형이 아니면, `TypeError`가 발생합니다.

O& (**object**) [*converter*, *address*]
converter 함수를 통해 파이썬 객체를 C 변수로 변환합니다. 두 개의 인자를 취합니다: 첫 번째는 함수이고, 두 번째는 `void*`로 변환된, (임의의 형의) C 변수의 주소입니다. *converter* 함수는 다음과 같이 호출됩니다:

```
status = converter(object, address);
```

여기서 *object*는 변환할 파이썬 객체이고 *address*는 `PyArg_Parse*` 함수에 전달된 `void*` 인자입니다. 반환된 *status*는 성공적인 변환의 경우 `1`이고 변환에 실패한 경우 `0`이어야 합니다. 변환이 실패하면, *converter* 함수는 예외를 발생시키고 *address*의 내용을 수정하지 않은 상태로 두어야 합니다. *converter*가 `Py_CLEANUP_SUPPORTED`를 반환하면, 인자 구문 분석이 결국 실패하면 두 번째로 호출되어 변환기에 이미 할당된 메모리를 해제할 기회를 제공할 수 있습니다. 이 두 번째 호출에서, *object* 매개변수는 `NULL`이 됩니다; *address*는 원래 호출과 같은 값을 갖습니다.

Examples of converters: *PyUnicode_FSConverter()* and *PyUnicode_FSDecoder()*.

버전 3.1에서 변경: `Py_CLEANUP_SUPPORTED`가 추가되었습니다.

p (**bool**) [**int**]
전달된 값의 논리값을 테스트(불리언 **p**redicate)하고 결과를 동등한 C 참/거짓 정숫값으로 변환합니다. 표현식이 참이면 int를 `1`로, 거짓이면 `0`으로 설정합니다. 모든 유효한 파이썬 값을 허용합니다. 파이썬이 논리값을 테스트하는 방법에 대한 자세한 내용은 truth를 참조하십시오.

Added in version 3.3.

(items) (**sequence**) [*matching-items*]
The object must be a Python sequence (except `str`, `bytes` or `bytearray`) whose length is the number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

If *items* contains format units which store a *borrowed buffer* (`s`, `s#`, `z`, `z#`, `y`, or `y#`) or a *borrowed reference* (`S`, `Y`, `U`, `O`, or `O!`), the object must be a Python tuple. The *converter* for the `O&` format unit in *items* must not store a borrowed buffer or a borrowed reference.

버전 3.14에서 변경: `str` and `bytearray` objects no longer accepted as a sequence.

버전 3.14부터 폐지됨: Non-tuple sequences are deprecated if *items* contains format units which store a borrowed buffer or a borrowed reference.

몇 가지 다른 문자는 포맷 문자열에서 의미가 있습니다. 중첩된 괄호 안에서는 나타날 수 없습니다. 그들은:

|
파이썬 인자 리스트의 나머지 인자가 선택 사항임을 나타냅니다. 선택적 인자에 해당하는 C 변수는 기본값으로 초기화되어야 합니다 — 선택적 인자가 지정되지 않을 때, *PyArg_ParseTuple()*은 해당 C 변수의 내용을 건드리지 않습니다.

$
PyArg_ParseTupleAndKeywords() 전용: 파이썬 인자 리스트의 나머지 인자가 키워드 전용임을

나타냅니다. 현재, 모든 키워드 전용 인자는 선택적 인자여야 하므로, | 는 항상 포맷 문자열에서 $ 앞에 지정되어야 합니다.

Added in version 3.3.

:
포맷 단위 리스트는 여기에서 끝납니다; 콜론 뒤의 문자열은 에러 메시지에서 함수 이름으로 사용됩니다 (*PyArg_ParseTuple()* 이 발생시키는 예외의 "연관된 값").

;
포맷 단위 리스트는 여기에서 끝납니다; 세미콜론 뒤의 문자열은 기본 에러 메시지의 에러 메시지 대신 에러 메시지로 사용됩니다. : 와 ; 는 서로를 배제합니다.

호출자에게 제공되는 모든 파이썬 객체 참조는 빌려온(*borrowed*) 참조임에 유의하십시오; 해제하지 마십시오 (즉, 참조 횟수를 줄이지 마십시오)!

이러한 함수에 전달되는 추가 인자는 포맷 문자열에 의해 형이 결정되는 변수의 주소여야 합니다; 이들은 입력 튜플의 값을 저장하는 데 사용됩니다. 위의 포맷 단위 리스트에서 설명된 대로, 이러한 매개 변수가 입력값으로 사용되는 몇 가지 경우가 있습니다; 이 경우 해당 포맷 단위에 대해 지정된 것과 일치해야 합니다.

변환이 성공하려면, *arg* 객체가 포맷과 일치해야 하며 포맷이 소진되어야 합니다. 성공하면, `PyArg_Parse*` 함수는 참을 반환하고, 그렇지 않으면 거짓을 반환하고 적절한 예외를 발생시킵니다. 포맷 단위 중 하나의 변환 실패로 인해 `PyArg_Parse*` 함수가 실패하면, 해당 주소의 변수와 그 뒤에 오는 포맷 단위는 건드리지 않습니다.

API 함수

int **PyArg_ParseTuple**(*PyObject* *args, const char *format, ...)

Part of the 안정 ABI. 위치 매개 변수만 지역 변수로 취하는 함수의 매개 변수를 구문 분석합니다. 성공하면 참을 반환합니다; 실패하면, 거짓을 반환하고 적절한 예외를 발생시킵니다.

int **PyArg_VaParse**(*PyObject* *args, const char *format, va_list vargs)

Part of the 안정 ABI. 가변 개수의 인자가 아닌 va_list를 받아들인다는 점을 제외하면, *PyArg_ParseTuple()* 과 동일합니다.

int **PyArg_ParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, ...)

Part of the 안정 ABI. 위치와 키워드 매개 변수를 모두 지역 변수로 취하는 함수의 매개 변수를 구문 분석합니다. *keywords* 인자는 널-종료된 ASCII 나 UTF-8 인코딩된 C 문자열로 지정된 키워드 매개 변수 이름의 `NULL`-종료 배열입니다. 빈 이름은 위치-전용 매개 변수를 나타냅니다. 성공하면 참을 반환합니다; 실패하면, 거짓을 반환하고 적절한 예외를 발생시킵니다.

> **참고**
>
> The *keywords* parameter declaration is `char *const*` in C and `const char *const*` in C++. This can be overridden with the *PY_CXX_CONST* macro.

버전 3.6에서 변경: 위치-전용 매개 변수에 대한 지원이 추가되었습니다.

버전 3.13에서 변경: The *keywords* parameter has now type `char *const*` in C and `const char *const*` in C++, instead of `char**`. Added support for non-ASCII keyword parameter names.

int **PyArg_VaParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, va_list vargs)

Part of the 안정 ABI. 가변 개수의 인자가 아닌 va_list를 받아들인다는 점을 제외하면, *PyArg_ParseTupleAndKeywords()* 와 동일합니다.

int **PyArg_ValidateKeywordArguments**(*PyObject**)

Part of the 안정 ABI. 키워드 인자 딕셔너리의 키가 문자열인지 확인합니다. *PyArg_ParseTupleAndKeywords()* 가 사용되지 않는 경우에만 필요합니다, 여기서는 이미 이 검사를 수행하기 때문입니다.

Added in version 3.2.

int **PyArg_Parse**(*PyObject* *args, const char *format, ...)

Part of the 안정 ABI. 단일 위치 매개 변수를 취하는 함수의 매개 변수를 지역 변수로 구문 분석합니다. 성공하면 참을 반환합니다; 실패하면, 거짓을 반환하고 적절한 예외를 발생시킵니다.

예제:

```
// METH_O 호출 규칙을 사용하는 함수
static PyObject*
my_function(PyObject *module, PyObject *arg)
{
    int value;
    if (!PyArg_Parse(arg, "i:my_function", &value)) {
        return NULL;
    }
    // ... value 를 사용합니다 ...
}
```

int **PyArg_UnpackTuple**(*PyObject* *args, const char *name, *Py_ssize_t* min, *Py_ssize_t* max, ...)

Part of the 안정 ABI. 인자 형을 지정하기 위해 포맷 문자열을 사용하지 않는 더 간단한 형식의 매개 변수 조회. 이 방법으로 매개 변수를 꺼내는 함수는 함수나 메서드 테이블에서 *METH_VARARGS*로 선언되어야 합니다. 실제 매개 변수를 포함하는 튜플은 *args*로 전달되어야 합니다; 이것은 실제로 튜플이어야 합니다. 튜플의 길이는 *min* 이상 *max* 이하이어야 합니다; *min*과 *max*는 같을 수 있습니다. 추가 인자는 함수에 전달되어야 하며, 각 인자는 *PyObject** 변수에 대한 포인터여야 합니다; 이들은 *args*의 값으로 채워집니다; 빌린 참조가 포함됩니다. *args*에서 제공하지 않는 선택적 매개 변수에 해당하는 변수는 채워지지 않습니다; 이것들은 호출자에 의해 초기화되어야 합니다. 이 함수는 성공하면 참을 반환하고 *args*가 튜플이 아니거나, 잘못된 수의 요소를 포함하면 거짓을 반환합니다; 실패하면 예외가 설정됩니다.

다음은 이 함수 사용의 예입니다, 약한 참조를 위한 `_weakref` 도우미 모듈의 소스에서 가져왔습니다:

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

이 예제에서 *PyArg_UnpackTuple()*에 대한 호출은 *PyArg_ParseTuple()*에 대한 다음 호출과 전적으로 동등합니다:

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

PY_CXX_CONST

The value to be inserted, if any, before `char *const*` in the *keywords* parameter declaration of *PyArg_ParseTupleAndKeywords()* and *PyArg_VaParseTupleAndKeywords()*. Default empty for C and `const` for C++ (`const char *const*`). To override, define it to the desired value before including `Python.h`.

Added in version 3.13.

7.6.2 값 구축

PyObject ***Py_BuildValue**(const char *format, ...)

반환값: 새 참조. *Part of the* 안정 ABI. `PyArg_Parse*` 함수 계열에서 받아들이는 것과 유사한 포맷 문자열과 값 시퀀스를 기반으로 새 값을 만듭니다. 값을 반환하거나 에러가 발생하면 `NULL`을 반환합니다; `NULL`이 반환되면 예외가 발생합니다.

*`Py_BuildValue()`*는 항상 튜플을 구축하지는 않습니다. 포맷 문자열에 둘 이상의 포맷 단위가 포함되었을 때만 튜플을 구축합니다. 포맷 문자열이 비어 있으면, `None`을 반환합니다; 정확히 하나의 포맷 단위를 포함하면, 해당 포맷 단위가 기술하는 객체가 무엇이건 반환합니다. 크기가 0이나 1인 튜플을 반환하도록 하려면, 포맷 문자열을 괄호로 묶으십시오.

`s`와 `s#` 포맷의 경우처럼, 메모리 버퍼가 데이터를 빌드 객체에 제공하기 위해 매개 변수로 전달될 때, 필요한 데이터가 복사됩니다. 호출자가 제공하는 버퍼는 *`Py_BuildValue()`*가 만든 객체에 의해 참조되지 않습니다. 즉, 여러분의 코드가 `malloc()`을 호출하고 할당된 메모리를 *`Py_BuildValue()`*에 전달하면, 인단 *`Py_BuildValue()`*가 반환되면 여러분이 코드가 해당 메모리에 대해 `free()`를 호출해야 합니다.

다음 설명에서, 인용된 형식은 포맷 단위입니다; (둥근) 괄호 안의 항목은 포맷 단위가 반환할 파이썬 객체 형입니다; [대괄호] 안의 항목은 전달할 C 값의 형입니다.

문자 스페이스, 탭, 콜론 및 쉼표는 포맷 문자열에서 무시됩니다 (하지만 `s#`와 같은 포맷 단위 내에서는 아닙니다). 이것은 긴 포맷 문자열을 좀 더 읽기 쉽게 만드는 데 사용할 수 있습니다.

`s` (**`str`** 또는 **`None`**) [**const char ***]
: `'utf-8'` 인코딩을 사용하여 널-종료 C 문자열을 파이썬 `str` 객체로 변환합니다. C 문자열 포인터가 `NULL`이면, `None`이 사용됩니다.

`s#` (**`str`** 또는 **`None`**) [**const char ***, ***Py_ssize_t***]
: `'utf-8'` 인코딩을 사용하여 C 문자열과 그 길이를 파이썬 `str` 객체로 변환합니다. C 문자열 포인터가 `NULL`이면, 길이가 무시되고 `None`이 반환됩니다.

`y` (**`bytes`**) [**const char ***]
: 이것은 C 문자열을 파이썬 `bytes` 객체로 변환합니다. C 문자열 포인터가 `NULL`이면, `None`이 반환됩니다.

`y#` (**`bytes`**) [**const char ***, ***Py_ssize_t***]
: 이것은 C 문자열과 그 길이를 파이썬 객체로 변환합니다. C 문자열 포인터가 `NULL`이면, `None`이 반환됩니다.

`z` (**`str`** 또는 **`None`**) [**const char ***]
: `s`와 같습니다.

`z#` (**`str`** 또는 **`None`**) [**const char ***, ***Py_ssize_t***]
: `s#`과 같습니다.

`u` (**`str`**) [**const wchar_t ***]
: 유니코드 (UTF-16 또는 UCS-4) 데이터의 널-종료 `wchar_t` 버퍼를 파이썬 유니코드 객체로 변환합니다. 유니코드 버퍼 포인터가 `NULL`이면, `None`이 반환됩니다.

`u#` (**`str`**) [**const wchar_t ***, ***Py_ssize_t***]
: 유니코드 (UTF-16 또는 UCS-4) 데이터 버퍼와 그 길이를 파이썬 유니코드 객체로 변환합니다. 유니코드 버퍼 포인터가 `NULL`이면, 길이가 무시되고 `None`이 반환됩니다.

`U` (**`str`** 또는 **`None`**) [**const char ***]
: `s`와 같습니다.

`U#` (**`str`** 또는 **`None`**) [**const char ***, ***Py_ssize_t***]
: `s#`과 같습니다.

`i` (**`int`**) [**int**]
: 일반 C `int`를 파이썬 정수 객체로 변환합니다.

`b` (**`int`**) [**char**]
: 일반 C `char`을 파이썬 정수 객체로 변환합니다.

h (int) [short int]
일반 C `short int`를 파이썬 정수 객체로 변환합니다.

l (int) [long int]
C `long int`를 파이썬 정수 객체로 변환합니다.

B (int) [unsigned char]
C `unsigned char`을 파이썬 정수 객체로 변환합니다.

H (int) [unsigned short int]
C `unsigned short int`를 파이썬 정수 객체로 변환합니다.

I (int) [unsigned int]
C `unsigned int`를 파이썬 정수 객체로 변환합니다.

k (int) [unsigned long]
C `unsigned long`을 파이썬 정수 객체로 변환합니다.

L (int) [long long]
C `long long`을 파이썬 정수 객체로 변환합니다.

K (int) [unsigned long long]
C `unsigned long long`을 파이썬 정수 객체로 변환합니다.

n (int) [*Py_ssize_t*]
C *Py_ssize_t*를 파이썬 정수로 변환합니다.

p (bool) [int]
Convert a C `int` to a Python `bool` object.

Be aware that this format requires an `int` argument. Unlike most other contexts in C, variadic arguments are not coerced to a suitable type automatically. You can convert another type (for example, a pointer or a float) to a suitable `int` value using `(x) ? 1 : 0` or `!!x`.

Added in version 3.14.

c (길이 1의 bytes) [char]
바이트를 나타내는 C `int`를 길이 1의 파이썬 `bytes` 객체로 변환합니다.

C (길이 1의 str) [int]
문자를 나타내는 C `int`를 길이 1의 파이썬 `str` 객체로 변환합니다.

d (float) [double]
C `double`을 파이썬 부동 소수점 숫자로 변환합니다.

f (float) [float]
C `float`를 파이썬 부동 소수점 숫자로 변환합니다.

D (complex) [Py_complex *]
C *Py_complex* 구조체를 파이썬 복소수로 변환합니다.

O (object) [PyObject *]
파이썬 객체를 그대로 전달하지만, 그 객체에 새로운 강한 참조를 만듭니다 (즉, 참조 횟수가 1 증가합니다). 전달된 객체가 `NULL` 포인터면, 인자를 생성하는 호출이 에러를 발견하고 예외를 설정했기 때문으로 간주합니다. 따라서, *Py_BuildValue()*는 `NULL`을 반환하지만, 예외를 발생시키지 않습니다. 아직 예외가 발생하지 않았으면, `SystemError`가 설정됩니다.

S (object) [PyObject *]
`O`와 같습니다.

N (object) [PyObject *]
`O`와 같지만, 새로운 강한 참조를 만들지 않습니다. 인자 리스트에서 객체 생성자를 호출하여 객체를 만들 때 유용합니다.

O& (object) [*converter*, *anything*]
converter 함수를 통해 *anything*을 파이썬 객체로 변환합니다. 함수는 인자로 *anything*(`void*`와 호환되어야 합니다)을 사용하여 호출되며 "새" 파이썬 객체를 반환하거나, 에러가 발생하면 `NULL`을 반환해야 합니다.

(items) (tuple) [*matching-items*]
C값의 시퀀스를 항목 수가 같은 파이썬 튜플로 변환합니다.

[items] (list) [*matching-items*]
C값의 시퀀스를 항목 수가 같은 파이썬 리스트로 변환합니다.

{items} (dict) [*matching-items*]
C값의 시퀀스를 파이썬 딕셔너리로 변환합니다. 연속된 C 값의 각 쌍은 딕셔너리에 하나의 항목을 추가하여, 각각 키와 값으로 사용됩니다.

포맷 문자열에 에러가 있으면, `SystemError` 예외가 설정되고 `NULL`이 반환됩니다.

PyObject ***Py_VaBuildValue** (const char *format, va_list vargs)
반환값: 새 참조. *Part of the* 안정 ABI. 가변 개수의 인자가 아닌 va_list를 받아들인다는 점을 제외하면, *Py_BuildValue()*와 동일합니다.

7.7 문자열 변환과 포매팅

숫자 변환과 포맷된 문자열 출력을 위한 함수.

int **PyOS_snprintf** (char *str, size_t size, const char *format, ...)
Part of the 안정 ABI. 포맷 문자열 *format*과 추가 인자에 따라 *size* 바이트를 넘지 않도록 *str*로 출력합니다. 유닉스 매뉴얼 페이지 *snprintf(3)*를 보십시오.

int **PyOS_vsnprintf** (char *str, size_t size, const char *format, va_list va)
Part of the 안정 ABI. 포맷 문자열 *format*과 가변 인자 목록 *va*에 따라 *size* 바이트를 넘지 않도록 *str*로 출력합니다. 유닉스 매뉴얼 페이지 *vsnprintf(3)*를 보십시오.

*PyOS_snprintf()*와 *PyOS_vsnprintf()*는 표준 C 라이브러리 함수 `snprintf()`와 `vsnprintf()`를 감쌉니다. 그들의 목적은 경계 조건에서 표준 C 함수가 제공하지 않는 수준의 일관된 동작을 보장하는 것입니다.

래퍼는 반환 시 `str[size-1]`이 항상 `'\0'`이 되도록 합니다. str에 *size* 바이트(후행 `'\0'` 포함)를 초과해서 쓰지 않습니다. 두 함수 모두 `str != NULL`, `size > 0`, `format != NULL` 및 `size < INT_MAX`를 요구합니다. 이는 필요한 버퍼 크기를 결정하는 C99 `n = snprintf(NULL, 0, ...)`에 해당하는 것이 없다는 뜻임에 유의하십시오.

이 함수들의 반환 값(*rv*)은 다음과 같이 해석되어야 합니다:

- `0 <= rv < size` 일 때, 출력 변환에 성공했으며 *rv* 문자가 *str*에 기록되었습니다 (`str[rv]`의 후행 `'\0'` 바이트 제외).
- `rv >= size` 일 때, 출력 변환이 잘렸고 성공하려면 `rv + 1` 바이트의 버퍼가 필요합니다. `str[size-1]`은 이때 `'\0'`입니다.
- `rv < 0` 일 때, "뭔가 나쁜 일이 일어났습니다." 이때도 `str[size-1]`은 `'\0'`이지만, *str*의 나머지는 정의되지 않습니다. 에러의 정확한 원인은 하부 플랫폼에 따라 다릅니다.

다음 함수는 로케일 독립적인 문자열에서 숫자로의 변환을 제공합니다.

unsigned long **PyOS_strtoul** (const char *str, char **ptr, int base)
Part of the 안정 ABI. Convert the initial part of the string in `str` to an `unsigned long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Leading white space and case of characters are ignored. If `base` is zero it looks for a leading `0b`, `0o` or `0x` to tell which base. If these are absent it defaults to `10`. Base must be 0 or between 2 and 36 (inclusive). If `ptr` is non-`NULL` it will contain a pointer to the end of the scan.

If the converted value falls out of range of corresponding return type, range error occurs (`errno` is set to `ERANGE`) and `ULONG_MAX` is returned. If no conversion can be performed, `0` is returned.

See also the Unix man page *strtoul(3)*.

Added in version 3.2.

long **PyOS_strtol**(const char *str, char **ptr, int base)

Part of the 안정 ABI. Convert the initial part of the string in `str` to an `long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Same as *PyOS_strtoul()*, but return a `long` value instead and `LONG_MAX` on overflows.

See also the Unix man page *strtol(3)*.

Added in version 3.2.

double **PyOS_string_to_double**(const char *s, char **endptr, *PyObject* *overflow_exception)

Part of the 안정 ABI. 문자열 `s`를 `double`로 변환하고, 실패 시 파이썬 예외를 발생시킵니다. 허용되는 문자열 집합은 `s`가 선행이나 후행 공백을 가질 수 없다는 점을 제외하고는 파이썬의 `float()` 생성자가 허용하는 문자열 집합에 대응합니다. 변환은 현재 로케일과 독립적입니다.

`endptr`이 `NULL`이면, 전체 문자열을 변환합니다. 문자열이 부동 소수점 숫자의 유효한 표현이 아니면 `ValueError`를 발생시키고 `-1.0`을 반환합니다.

endptr이 `NULL`이 아니면, 가능한 한 많은 문자열을 변환하고 `*endptr`이 변환되지 않은 첫 번째 문자를 가리키도록 설정합니다. 문자열의 초기 세그먼트가 부동 소수점 숫자의 유효한 표현이 아니면, `*endptr`이 문자열의 시작을 가리키도록 설정하고, ValueError를 발생시키고 `-1.0`을 반환합니다.

If `s` represents a value that is too large to store in a float (for example, `"1e500"` is such a string on many platforms) then if `overflow_exception` is `NULL` return `Py_INFINITY` (with an appropriate sign) and don't set any exception. Otherwise, `overflow_exception` must point to a Python exception object; raise that exception and return `-1.0`. In both cases, set `*endptr` to point to the first character after the converted value.

변환 중 다른 에러가 발생하면 (예를 들어 메모리 부족 에러), 적절한 파이썬 예외를 설정하고 `-1.0`을 반환합니다.

Added in version 3.1.

char ***PyOS_double_to_string**(double val, char format_code, int precision, int flags, int *ptype)

Part of the 안정 ABI. 제공된 *format_code*, *precision* 및 *flags*를 사용하여 `double` *val*을 문자열로 변환합니다.

*format_code*는 `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` 또는 `'r'` 중 하나여야 합니다. `'r'`의 경우, 제공된 *precision*은 0이어야 하며 무시됩니다. `'r'` 포맷 코드는 표준 `repr()` 형식을 지정합니다.

*flags*는 `Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0` 또는 `Py_DTSF_ALT` 값을 0개 이상 함께 or 할 수 있습니다:

- `Py_DTSF_SIGN`은 *val*가 음수가 아닐 때도 항상 반환된 문자열 앞에 부호 문자가 오는 것을 뜻합니다.
- `Py_DTSF_ADD_DOT_0`은 반환된 문자열이 정수처럼 보이지 않도록 하는 것을 뜻합니다.
- `Py_DTSF_ALT`는 "대체" 포매팅 규칙을 적용하는 것을 뜻합니다. 자세한 내용은 *PyOS_snprintf()* `'#'` 지정자에 대한 설명서를 참조하십시오.

*ptype*이 `NULL`이 아니면, 포인터가 가리키는 값은 `Py_DTST_FINITE`, `Py_DTST_INFINITE` 또는 `Py_DTST_NAN` 중 하나로 설정되어, *val*가 각각 유한 수, 무한 수 또는 NaN임을 나타냅니다.

반환 값은 변환된 문자열이 있는 *buffer*에 대한 포인터이거나, 변환에 실패하면 `NULL`입니다. 호출자는 *PyMem_Free()*를 호출하여 반환된 문자열을 해제해야 합니다.

Added in version 3.1.

int **PyOS_stricmp**(const char *s1, const char *s2)

대소 문자 구분 없는 문자열 비교. 이 함수는 대소 문자를 무시한다는 점만 제외하면 `strcmp()`와 거의 같게 작동합니다.

int **PyOS_strnicmp**(const char *s1, const char *s2, *Py_ssize_t* size)

대소 문자 구분 없는 문자열 비교. 이 함수는 대소 문자를 무시한다는 점만 제외하면 `strncmp()`와 거의 같게 작동합니다.

7.8 PyHash API

See also the *PyTypeObject.tp_hash* member and numeric-hash.

type **Py_hash_t**

Hash value type: signed integer.

Added in version 3.2.

type **Py_uhash_t**

Hash value type: unsigned integer.

Added in version 3.2.

PyHASH_MODULUS

The Mersenne prime `P = 2**n -1`, used for numeric hash scheme.

Added in version 3.13.

PyHASH_BITS

The exponent `n` of `P` in *PyHASH_MODULUS*.

Added in version 3.13.

PyHASH_MULTIPLIER

Prime multiplier used in string and various other hashes.

Added in version 3.13.

PyHASH_INF

The hash value returned for a positive infinity.

Added in version 3.13.

PyHASH_IMAG

The multiplier used for the imaginary part of a complex number.

Added in version 3.13.

type **PyHash_FuncDef**

Hash function definition used by *PyHash_GetFuncDef()*.

Py_hash_t (*const **hash**)(const void*, *Py_ssize_t*)

Hash function.

const char ***name**

Hash function name (UTF-8 encoded string).

const int **hash_bits**

Internal size of the hash value in bits.

const int **seed_bits**

Size of seed input in bits.

Added in version 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

Get the hash function definition.

> **더 보기**
>
> **PEP 456** "Secure and interchangeable hash algorithm".

Added in version 3.4.

Py_hash_t **Py_HashPointer**(const void *ptr)

Hash a pointer value: process the pointer value as an integer (cast it to `uintptr_t` internally). The pointer is not dereferenced.

The function cannot fail: it cannot return `-1`.

Added in version 3.13.

Py_hash_t **Py_HashBuffer**(const void *ptr, *Py_ssize_t* len)

Compute and return the hash value of a buffer of *len* bytes starting at address *ptr*. The hash is guaranteed to match that of `bytes`, `memoryview`, and other built-in objects that implement the *buffer protocol*.

Use this function to implement hashing for immutable objects whose *tp_richcompare* function compares to another object's buffer.

len must be greater than or equal to `0`.

This function always succeeds.

Added in version 3.14.

Py_hash_t **PyObject_GenericHash**(*PyObject* *obj)

Generic hashing function that is meant to be put into a type object's `tp_hash` slot. Its result only depends on the object's identity.

CPython 구현 상세: In CPython, it is equivalent to *Py_HashPointer()*.

Added in version 3.13.

7.9 리플렉션

PyObject ***PyEval_GetBuiltins**(void)

반환값: 빌린 참조. *Part of the* 안정 ABI. 버전 3.13부터 폐지됨: 대신 *PyEval_GetFrameBuiltins()*를 사용하십시오.

현재 실행 프레임이나 현재 실행 중인 프레임이 없으면 스레드 상태의 인터프리터의 builtins의 딕셔너리를 반환합니다.

PyObject ***PyEval_GetLocals**(void)

반환값: 빌린 참조. *Part of the* 안정 ABI. 버전 3.13부터 폐지됨: Use either *PyEval_GetFrameLocals()* to obtain the same behaviour as calling `locals()` in Python code, or else call *PyFrame_GetLocals()* on the result of *PyEval_GetFrame()* to access the `f_locals` attribute of the currently executing frame.

현재 실행 프레임의 지역 변수로의 접근을 제공하는 매핑을 반환하거나, 현재 실행 중인 프레임이 없으면 `NULL`을 반환합니다.

Refer to `locals()` for details of the mapping returned at different scopes.

As this function returns a *borrowed reference*, the dictionary returned for *optimized scopes* is cached on the frame object and will remain alive as long as the frame object does. Unlike *PyEval_GetFrameLocals()* and `locals()`, subsequent calls to this function in the same frame will update the contents of the cached dictionary to reflect changes in the state of the local variables rather than returning a new snapshot.

버전 3.13에서 변경: As part of **PEP 667**, *PyFrame_GetLocals()*, `locals()`, and `FrameType.f_locals` no longer make use of the shared cache dictionary. Refer to the What's New entry for additional details.

PyObject ***PyEval_GetGlobals**(void)

반환값: 빌린 참조. *Part of the* 안정 ABI. 버전 3.13부터 폐지됨: 대신 *PyEval_GetFrameGlobals()*를 사용하십시오.

현재 실행 프레임의 전역 변수 딕셔너리를 반환하거나, 현재 실행 중인 프레임이 없으면 `NULL`을 반환합니다.

PyFrameObject ***PyEval_GetFrame**(void)

반환값: 빌린 참조. *Part of the* 안정 ABI. Return the *attached thread state*' s frame, which is `NULL` if no frame is currently executing.

*`PyThreadState_GetFrame()`*도 참조하십시오.

PyObject ***PyEval_GetFrameBuiltins**(void)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.13* 이후로. 현재 실행 프레임이나 현재 실행 중인 프레임이 없으면 스레드 상태의 인터프리터의 builtins의 딕셔너리를 반환합니다.

Added in version 3.13.

PyObject ***PyEval_GetFrameLocals**(void)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.13* 이후로. 현재 실행 프레임의 지역 변수 딕셔너리를 반환하거나, 현재 실행 중인 프레임이 없으면 `NULL`을 반환합니다. 파이썬 코드에서 `locals()`를 호출하는 것과 동등합니다.

To access `f_locals` on the current frame without making an independent snapshot in *optimized scopes*, call *`PyFrame_GetLocals()`* on the result of *`PyEval_GetFrame()`*.

Added in version 3.13.

PyObject ***PyEval_GetFrameGlobals**(void)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.13* 이후로. 현재 실행 프레임의 전역 변수 딕셔너리를 반환하거나, 현재 실행 중인 프레임이 없으면 `NULL`을 반환합니다. 파이썬 코드에서 `globals()`를 호출하는 것과 동등합니다.

Added in version 3.13.

const char ***PyEval_GetFuncName**(*PyObject* *func)

Part of the 안정 ABI. *func*가 함수, 클래스 또는 인스턴스 객체면 *func*의 이름을 반환하고, 그렇지 않으면 *func*의 형의 이름을 반환합니다.

const char ***PyEval_GetFuncDesc**(*PyObject* *func)

Part of the 안정 ABI. *func*의 형에 따라 설명 문자열을 반환합니다. 반환 값에는 함수 및 메서드의 "()", " constructor", " instance" 및 " object" 가 포함됩니다. *`PyEval_GetFuncName()`*의 결과와 이어붙이면 *func*의 설명이 됩니다.

7.10 코덱 등록소와 지원 함수

int **PyCodec_Register**(*PyObject* *search_function)

Part of the 안정 ABI. 새로운 코덱 검색 함수를 등록합니다.

부작용으로, 아직 로드되지 않았다면, `encodings` 패키지를 로드하여 항상 검색 함수 목록의 첫 번째 항목이 되도록 합니다.

int **PyCodec_Unregister**(*PyObject* *search_function)

Part of the 안정 ABI 버전 *3.10* 이후로. Unregister a codec search function and clear the registry' s cache. If the search function is not registered, do nothing. Return 0 on success. Raise an exception and return -1 on error.

Added in version 3.10.

int **PyCodec_KnownEncoding**(const char *encoding)

Part of the 안정 ABI. 지정된 *encoding*에 대해 등록된 코덱이 있는지에 따라 `1` 이나 `0`을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyCodec_Encode**(*PyObject* *object, const char *encoding, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 일반 코덱 기반 인코딩 API.

*object*는 *errors*로 정의된 에러 처리 방법을 사용하여 지정된 *encoding*에 대해 발견된 인코더 함수로 전달됩니다. 코덱에 정의된 기본 방법을 사용하기 위해 *errors*가 `NULL` 일 수 있습니다. 인코더를 찾을 수 없으면 `LookupError`를 발생시킵니다.

PyObject ***PyCodec_Decode**(*PyObject* *object, const char *encoding, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 일반 코덱 기반 디코딩 API.

*object*는 *errors*로 정의된 에러 처리 방법을 사용하여 지정된 *encoding*에 대해 발견된 디코더 함수로 전달됩니다. 코덱에 정의된 기본 방법을 사용하기 위해 *errors*가 `NULL` 일 수 있습니다. 인코더를 찾을 수 없으면 `LookupError`를 발생시킵니다.

7.10.1 코덱 조회 API

다음 함수에서, *encoding* 문자열은 모두 소문자로 변환되어 조회되므로, 이 메커니즘을 통한 인코딩 조회는 대소문자를 구분하지 않게 됩니다. 코덱이 없으면, `KeyError`가 설정되고 `NULL`이 반환됩니다.

PyObject ***PyCodec_Encoder**(const char *encoding)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 *encoding*에 대한 인코더 함수를 가져옵니다.

PyObject ***PyCodec_Decoder**(const char *encoding)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 *encoding*에 대한 디코더 함수를 가져옵니다.

PyObject ***PyCodec_IncrementalEncoder**(const char *encoding, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 지정된 *encoding*에 대한 `IncrementalEncoder` 객체를 가져옵니다.

PyObject ***PyCodec_IncrementalDecoder**(const char *encoding, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 지정된 *encoding*에 대한 `IncrementalDecoder` 객체를 가져옵니다.

PyObject ***PyCodec_StreamReader**(const char *encoding, *PyObject* *stream, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 지정된 *encoding*에 대한 `StreamReader` 팩토리 함수를 가져옵니다.

PyObject ***PyCodec_StreamWriter**(const char *encoding, *PyObject* *stream, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 지정된 *encoding*에 대한 `StreamWriter` 팩토리 함수를 가져옵니다.

7.10.2 유니코드 인코딩 에러 처리기용 등록소 API

int **PyCodec_RegisterError**(const char *name, *PyObject* *error)

Part of the 안정 ABI. 지정된 *name* 으로 에러 처리 콜백 함수 *error*를 등록합니다. 코덱이 인코딩할 수 없는 문자/디코딩할 수 없는 바이트열을 발견하고, 인코드/디코드 함수를 호출할 때 *name*이 error 매개 변수로 지정되었을 때 이 콜백 함수를 호출합니다.

콜백은 하나의 인자로 `UnicodeEncodeError`, `UnicodeDecodeError` 또는 `UnicodeTranslateError`의 인스턴스를 받아들이는데, 문제가 되는 문자나 바이트의 시퀀스와 이들의 원본 문자열에서의 오프셋에 대한 정보를 담고 있습니다 (이 정보를 추출하는 함수는 유니코드 예외 객체를 참조하세요). 콜백은 주어진 예외를 발생시키거나, 문제가 있는 시퀀스의 대체와 원래 문자열에서 인코딩/디코딩을 다시 시작해야 하는 오프셋을 제공하는 정수를 포함하는 두 항목 튜플을 반환해야 합니다.

성공하면 `0`을, 에러면 `-1`을 반환합니다.

PyObject ***PyCodec_LookupError**(const char *name)

반환값: 새 참조. *Part of the* 안정 ABI. *name* 으로 등록된 에러 처리 콜백 함수를 찾습니다. 특수한 경우로 `NULL`이 전달 될 수 있는데, 이때는 "strict" 에 대한 에러 처리 콜백이 반환됩니다.

PyObject ***PyCodec_StrictErrors**(*PyObject* *exc)

반환값: 항상 *NULL*. *Part of the* 안정 ABI. *exc*를 예외로 발생시킵니다.

PyObject ***PyCodec_IgnoreErrors**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI. 잘못된 입력을 건너뛰고, 유니코드 에러를 무시합니다.

PyObject ***PyCodec_ReplaceErrors**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI. 유니코드 인코딩 에러를 `?` 나 `U+FFFD`로 치환합니다.

PyObject ***PyCodec_XMLCharRefReplaceErrors**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI. 유니코드 인코딩 에러를 XML 문자 참조로 치환합니다.

PyObject ***PyCodec_BackslashReplaceErrors**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI. 유니코드 인코딩 에러를 백 슬래시 이스케이프(`\x`, `\u` 및 `\U`)로 치환합니다.

PyObject ***PyCodec_NameReplaceErrors**(*PyObject* *exc)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. 유니코드 인코딩 에러를 `\N{...}` 이스케이프로 치환합니다.

Added in version 3.5.

7.11 PyTime C API

Added in version 3.13.

The clock C API provides access to system clocks. It is similar to the Python `time` module.

For C API related to the `datetime` module, see *DateTime 객체*.

7.11.1 Types

type **PyTime_t**

A timestamp or duration in nanoseconds, represented as a signed 64-bit integer.

The reference point for timestamps depends on the clock used. For example, *`PyTime_Time()`* returns timestamps relative to the UNIX epoch.

The supported range is around [-292.3 years; +292.3 years]. Using the Unix epoch (January 1st, 1970) as reference, the supported date range is around [1677-09-21; 2262-04-11]. The exact limits are exposed as constants:

PyTime_t **PyTime_MIN**

Minimum value of *`PyTime_t`*.

PyTime_t **PyTime_MAX**

Maximum value of *`PyTime_t`*.

7.11.2 Clock Functions

The following functions take a pointer to a *`PyTime_t`* that they set to the value of a particular clock. Details of each clock are given in the documentation of the corresponding Python function.

The functions return `0` on success, or `-1` (with an exception set) on failure.

On integer overflow, they set the *`PyExc_OverflowError`* exception and set `*result` to the value clamped to the `[PyTime_MIN; PyTime_MAX]` range. (On current systems, integer overflows are likely caused by misconfigured system time.)

As any other C API (unless otherwise specified), the functions must be called with an *attached thread state*.

int **PyTime_Monotonic**(*PyTime_t* *result)

Read the monotonic clock. See `time.monotonic()` for important details on this clock.

int **PyTime_PerfCounter**(*PyTime_t* *result)

Read the performance counter. See `time.perf_counter()` for important details on this clock.

int **PyTime_Time**(*PyTime_t* *result)

Read the “wall clock” time. See `time.time()` for details important on this clock.

7.11.3 Raw Clock Functions

Similar to clock functions, but don' t set an exception on error and don' t require the caller to have an *attached thread state*.

On success, the functions return `0`.

On failure, they set `*result` to `0` and return `-1`, *without* setting an exception. To get the cause of the error, *attach* a *thread state*, and call the regular (non-`Raw`) function. Note that the regular function may succeed after the `Raw` one failed.

int **PyTime_MonotonicRaw**(*PyTime_t* *result)

Similar to `PyTime_Monotonic()`, but don' t set an exception on error and don' t require an *attached thread state*.

int **PyTime_PerfCounterRaw**(*PyTime_t* *result)

Similar to `PyTime_PerfCounter()`, but don' t set an exception on error and don' t require an *attached thread state*.

int **PyTime_TimeRaw**(*PyTime_t* *result)

Similar to `PyTime_Time()`, but don' t set an exception on error and don' t require an *attached thread state*.

7.11.4 Conversion functions

double **PyTime_AsSecondsDouble**(*PyTime_t* t)

Convert a timestamp to a number of seconds as a C `double`.

The function cannot fail, but note that `double` has limited accuracy for large values.

7.12 Support for Perf Maps

On supported platforms (as of this writing, only Linux), the runtime can take advantage of *perf map files* to make Python functions visible to an external profiling tool (such as perf). A running process may create a file in the `/tmp` directory, which contains entries that can map a section of executable code to a name. This interface is described in the documentation of the Linux Perf tool.

In Python, these helper APIs can be used by libraries and features that rely on generating machine code on the fly.

Note that holding an *attached thread state* is not required for these APIs.

int **PyUnstable_PerfMapState_Init**(void)

이것은 불안정 *API*. It may change without warning in minor releases.

Open the `/tmp/perf-$pid.map` file, unless it' s already opened, and create a lock to ensure thread-safe writes to the file (provided the writes are done through `PyUnstable_WritePerfMapEntry()`). Normally, there' s no need to call this explicitly; just use `PyUnstable_WritePerfMapEntry()` and it will initialize the state on first call.

Returns `0` on success, `-1` on failure to create/open the perf map file, or `-2` on failure to create a lock. Check `errno` for more information about the cause of a failure.

int **PyUnstable_WritePerfMapEntry**(const void *code_addr, unsigned int code_size, const char *entry_name)

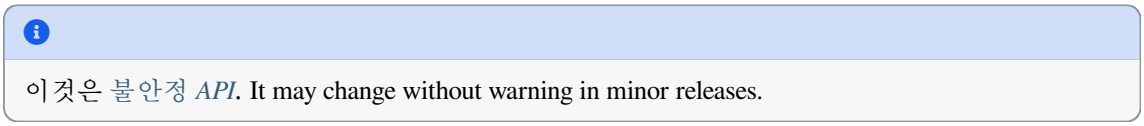
이것은 불안정 *API*. It may change without warning in minor releases.

Write one single entry to the `/tmp/perf-$pid.map` file. This function is thread safe. Here is what an example entry looks like:

```
# address      size  name
7f3529fcf759 b     py::bar:/run/t.py
```

Will call *`PyUnstable_PerfMapState_Init()`* before writing the entry, if the perf map file is not already opened. Returns `0` on success, or the same error codes as *`PyUnstable_PerfMapState_Init()`* on failure.

void **`PyUnstable_PerfMapState_Fini`**(void)



이것은 불안정 *API*. It may change without warning in minor releases.

Close the perf map file opened by *`PyUnstable_PerfMapState_Init()`*. This is called by the runtime itself during interpreter shut-down. In general, there shouldn' t be a reason to explicitly call this, except to handle specific scenarios such as forking.

CHAPTER 8

추상 객체 계층

이 장의 함수는 객체의 형과 무관하게, 혹은 광범위한 종류의 객체 형의 (예를 들어, 모든 숫자 형 또는 모든 시퀀스 형) 파이썬 객체와 상호 작용합니다. 적용되지 않는 객체 형에 사용되면, 파이썬 예외가 발생합니다.

*PyList_New()*로 만들었지만, 항목이 아직 NULL이 아닌 값으로 설정되지 않은 리스트 객체와 같이, 제대로 초기화되지 않은 객체에 대해서는 이 함수를 사용할 수 없습니다.

8.1 객체 프로토콜

PyObject ***Py_GetConstant**(unsigned int constant_id)

Part of the 안정 ABI 버전 *3.13* 이후로. Get a *strong reference* to a constant.

Set an exception and return NULL if *constant_id* is invalid.

constant_id must be one of these constant identifiers:

Constant Identifier	Value	Returned object
Py_CONSTANT_NONE	0	None
Py_CONSTANT_FALSE	1	False
Py_CONSTANT_TRUE	2	True
Py_CONSTANT_ELLIPSIS	3	Ellipsis
Py_CONSTANT_NOT_IMPLEMENTI	4	NotImplemented
Py_CONSTANT_ZERO	5	0
Py_CONSTANT_ONE	6	1
Py_CONSTANT_EMPTY_STR	7	''
Py_CONSTANT_EMPTY_BYTES	8	b''
Py_CONSTANT_EMPTY_TUPLE	9	()

Numeric values are only given for projects which cannot use the constant identifiers.

Added in version 3.13.

CPython 구현 상세: In CPython, all of these constants are *immortal*.

PyObject ***Py_GetConstantBorrowed**(unsigned int constant_id)

Part of the 안정 ABI 버전 *3.13* 이후로. Similar to *Py_GetConstant()*, but return a *borrowed reference*.

This function is primarily intended for backwards compatibility: using *Py_GetConstant()* is recommended for new code.

The reference is borrowed from the interpreter, and is valid until the interpreter finalization.

Added in version 3.13.

PyObject ***Py_NotImplemented**

지정된 형 조합에 대해 연산이 구현되지 않았음을 알리는 데 사용되는 NotImplemented 싱글톤.

Py_RETURN_NOTIMPLEMENTED

C 함수 내에서 *Py_NotImplemented* 반환을 올바르게 처리합니다 (즉, NotImplemented에 대한 새로운 강한 참조를 만들고 반환합니다).

Py_PRINT_RAW

Flag to be used with multiple functions that print the object (like *PyObject_Print()* and *PyFile_WriteObject()*). If passed, these function would use the str() of the object instead of the repr().

int **PyObject_Print**(*PyObject* *o, FILE *fp, int flags)

파일 *fp*에 객체 *o*를 인쇄합니다. 에러 시 `-1`을 반환합니다. flags 인자는 특정 인쇄 옵션을 활성화하는 데 사용됩니다. 현재 지원되는 유일한 옵션은 *Py_PRINT_RAW*입니다; 주어지면, `repr()` 대신 객체의 `str()`이 기록됩니다.

int **PyObject_HasAttrWithError**(*PyObject* *o, *PyObject* *attr_name)

Part of the 안정 ABI 버전 *3.13* 이후로. *o*에 *attr_name* 어트리뷰트가 있으면 `1`을, 그렇지 않으면 `0`을 반환합니다. 이것은 파이썬 표현식 `hasattr(o, attr_name)`과 동등합니다. 실패하면, `-1`을 반환합니다.

Added in version 3.13.

int **PyObject_HasAttrStringWithError**(*PyObject* *o, const char *attr_name)

Part of the 안정 ABI 버전 *3.13* 이후로. This is the same as *PyObject_HasAttrWithError()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Added in version 3.13.

int **PyObject_HasAttr**(*PyObject* *o, *PyObject* *attr_name)

Part of the 안정 ABI. *o*에 *attr_name* 어트리뷰트가 있으면 `1`을, 그렇지 않으면 `0`을 반환합니다. 이 함수는 항상 성공합니다.

> **참고**
>
> Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods aren't propagated, but instead given to `sys.unraisablehook()`. For proper error handling, use *PyObject_HasAttrWithError()*, *PyObject_GetOptionalAttr()* or *PyObject_GetAttr()* instead.

int **PyObject_HasAttrString**(*PyObject* *o, const char *attr_name)

Part of the 안정 ABI. This is the same as *PyObject_HasAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

> **참고**
>
> `__getattr__()`과 `__getattribute__()` 메서드를 호출할 때나 임시 `str` 객체를 만드는 중에 발생하는 예외는 조용히 무시됩니다. 적절한 에러 처리를 위해서는, 대신 *PyObject_HasAttrStringWithError()*, *PyObject_GetOptionalAttrString()* 또는 *PyObject_GetAttrString()*을 사용하십시오.

PyObject ***PyObject_GetAttr**(*PyObject* *o, *PyObject* *attr_name)

반환값: 새 참조. *Part of the* 안정 ABI. 객체 *o*에서 *attr_name*이라는 이름의 어트리뷰트를 가져옵니다. 성공하면 어트리뷰트 값을, 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o.attr_name`과 동등합니다.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttr()* instead.

PyObject ***PyObject_GetAttrString**(*PyObject* *o, const char *attr_name)

반환값: 새 참조. *Part of the* 안정 ABI. This is the same as *PyObject_GetAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttrString()* instead.

int **PyObject_GetOptionalAttr**(*PyObject* *obj, *PyObject* *attr_name, *PyObject* **result);

Part of the 안정 ABI 버전 *3.13* 이후로. Variant of *PyObject_GetAttr()* which doesn't raise `AttributeError` if the attribute is not found.

If the attribute is found, return `1` and set **result* to a new *strong reference* to the attribute. If the attribute is not found, return `0` and set **result* to `NULL`; the `AttributeError` is silenced. If an error other than `AttributeError` is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int **PyObject_GetOptionalAttrString**(*PyObject* *obj, const char *attr_name, *PyObject* **result);

Part of the 안정 ABI 버전 *3.13* 이후로. This is the same as `PyObject_GetOptionalAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

PyObject ***PyObject_GenericGetAttr**(*PyObject* *o, *PyObject* *name)

반환값: 새 참조. *Part of the* 안정 ABI. 형 객체의 `tp_getattro` 슬롯에 배치되는 일반 어트리뷰트 게터(getter) 함수. 객체의 (있다면) `__dict__`에 있는 어트리뷰트뿐만 아니라 객체의 MRO에 있는 클래스의 딕셔너리에 있는 디스크립터를 찾습니다. descriptors에 요약된 것처럼, 데이터 디스크립터는 인스턴스 어트리뷰트보다 우선하지만, 비 데이터 디스크립터는 그렇지 않습니다. 그렇지 않으면, `AttributeError`가 발생합니다.

int **PyObject_SetAttr**(*PyObject* *o, *PyObject* *attr_name, *PyObject* *v)

Part of the 안정 ABI. 객체 *o*에 대해, *attr_name*이라는 이름의 어트리뷰트 값을 *v* 값으로 설정합니다. 실패 시 예외를 발생시키고 `-1`을 반환합니다. 성공하면 `0`을 반환합니다. 이것은 파이썬 문장 `o.attr_name = v`와 동등합니다.

*v*가 `NULL`이면, 어트리뷰트가 삭제됩니다. 이 동작은 폐지되었고 `PyObject_DelAttr()`로 대체되었습니다만, 현재 제거할 계획이 없습니다.

int **PyObject_SetAttrString**(*PyObject* *o, const char *attr_name, *PyObject* *v)

Part of the 안정 ABI. This is the same as `PyObject_SetAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

*v*가 `NULL`이면, 어트리뷰트가 삭제되지만, 이 기능은 폐지되었고 `PyObject_DelAttrString()`으로 대체되었습니다.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_SetAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object.

int **PyObject_GenericSetAttr**(*PyObject* *o, *PyObject* *name, *PyObject* *value)

Part of the 안정 ABI. 형 객체의 `tp_setattro` 슬롯에 배치되는 일반 어트리뷰트 세터(setter)와 딜리터(deleter) 함수. 객체의 MRO에 있는 클래스의 딕셔너리에서 데이터 디스크립터를 찾고, 발견되면 인스턴스 딕셔너리에 있는 어트리뷰트를 설정하거나 삭제하는 것보다 우선합니다. 그렇지 않으면, 객체의 (있다면) `__dict__`에서 어트리뷰트가 설정되거나 삭제됩니다. 성공하면 `0`이 반환되고, 그렇지 않으면 `AttributeError`가 발생하고 `-1`이 반환됩니다.

int **PyObject_DelAttr**(*PyObject* *o, *PyObject* *attr_name)

Part of the 안정 ABI 버전 *3.13* 이후로. 객체 *o*에 대해, *attr_name*이라는 이름의 어트리뷰트를 삭제합니다. 실패 시 `-1`을 반환합니다. 이것은 파이썬 문장 `del o.attr_name`과 동등합니다.

int **PyObject_DelAttrString**(*PyObject* *o, const char *attr_name)

Part of the 안정 ABI 버전 *3.13* 이후로. This is the same as `PyObject_DelAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_DelAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object for lookup.

PyObject ***PyObject_GenericGetDict**(*PyObject* *o, void *context)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.10* 이후로. `__dict__` 디스크립터의 게터(getter)를 위한 일반적인 구현. 필요하면 딕셔너리를 만듭니다.

This function may also be called to get the `__dict__` of the object *o*. Pass `NULL` for *context* when calling it. Since this function may need to allocate memory for the dictionary, it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

On failure, returns `NULL` with an exception set.

Added in version 3.3.

int **PyObject_GenericSetDict**(*PyObject* *o, *PyObject* *value, void *context)

Part of the 안정 ABI 버전 *3.7* 이후로. `__dict__` 디스크립터의 세터(setter)를 위한 일반적인 구현. 이 구현은 딕셔너리 삭제를 허락하지 않습니다.

Added in version 3.3.

PyObject ***_PyObject_GetDictPtr**(*PyObject* *obj)

Return a pointer to `__dict__` of the object *obj*. If there is no `__dict__`, return `NULL` without setting an exception.

This function may need to allocate memory for the dictionary, so it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

PyObject ***PyObject_RichCompare**(*PyObject* *o1, *PyObject* *o2, int opid)

반환값: 새 참조. *Part of the* 안정 ABI. *opid*에 의해 지정된 연산을 사용하여 *o1*과 *o2*의 값을 비교합니다. *opid*는 `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT` 또는 `Py_GE` 중 하나여야 하고 각각 <, <=, ==, !=, > 또는 >=에 해당합니다. 이는 파이썬 표현식 `o1 op o2`와 동등합니다. 여기서 `op`는 *opid*에 해당하는 연산자입니다. 성공 시 비교 값을, 실패 시 `NULL`을 반환합니다.

int **PyObject_RichCompareBool**(*PyObject* *o1, *PyObject* *o2, int opid)

Part of the 안정 ABI. Compare the values of *o1* and *o2* using the operation specified by *opid*, like `PyObject_RichCompare()`, but returns `-1` on error, `0` if the result is false, `1` otherwise.

참고

*o1*과 *o2*가 같은 객체이면, `PyObject_RichCompareBool()`은 항상 `Py_EQ`의 경우는 `1`을, `Py_NE`의 경우는 `0`을 반환합니다.

PyObject ***PyObject_Format**(*PyObject* *obj, *PyObject* *format_spec)

Part of the 안정 ABI. Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

format_spec may be `NULL`. In this case the call is equivalent to `format(obj)`. Returns the formatted string on success, `NULL` on failure.

PyObject ***PyObject_Repr**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 객체 *o*의 문자열 표현을 계산합니다. 성공하면 문자열 표현을, 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `repr(o)`와 동등합니다. `repr()` 내장 함수에 의해 호출됩니다.

버전 3.4에서 변경: 이 함수에는 이제 디버그 어서션이 포함되어 있어 활성 예외를 조용히 버리지 않도록 합니다.

PyObject ***PyObject_ASCII**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. `PyObject_Repr()`처럼, 객체 *o*의 문자열 표현을 계산하지만, `\x`, `\u` 또는 `\U` 이스케이프를 사용하여 `PyObject_Repr()`이 반환한 문자열에서 비 ASCII 문자를 이스케이프 합니다. 이것은 파이썬 2에서 `PyObject_Repr()`에 의해 반환된 것과 유사한 문자열을 생성합니다. `ascii()` 내장 함수에 의해 호출됩니다.

PyObject ***PyObject_Str**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 객체 *o*의 문자열 표현을 계산합니다. 성공 시 문자열 표현을, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `str(o)`와 동등합니다. `str()` 내장 함수에 의해, 따라서 `print()` 함수에 의해서도 호출됩니다.

버전 3.4에서 변경: 이 함수에는 이제 디버그 어서션이 포함되어 있어 활성 예외를 조용히 버리지 않도록 합니다.

PyObject ***PyObject_Bytes**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 객체 *o*의 바이트열 표현을 계산합니다. 실패하면 NULL을, 성공하면 바이트열 객체를 반환됩니다. 이는 *o*가 정수가 아닐 때 파이썬 표현식 bytes(o)와 동등합니다. bytes(o)와 달리, *o*가 정수이면 0으로 초기화된 바이트열 객체 대신 TypeError가 발생합니다.

int **PyObject_IsSubclass**(*PyObject* *derived, *PyObject* *cls)

Part of the 안정 ABI. 클래스 *derived*가 클래스 *cls*와 동일하거나 *cls*에서 파생되었으면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 에러가 발생하면 -1을 반환합니다.

*cls*가 튜플이면, *cls*의 모든 항목에 대해 검사가 수행됩니다. 적어도 하나의 검사에서 1을 반환하면 결과는 1이 되고, 그렇지 않으면 0이 됩니다.

*cls*에 __subclasscheck__() 메서드가 있으면, **PEP 3119**에 설명된 대로 서브 클래스 상태를 판별하기 위해 호출됩니다. 그렇지 않으면, *derived*가 직접 또는 간접 서브 클래스일 때 *cls*의 서브 클래스입니다, 즉 cls.__mro__에 포함되어 있습니다.

일반적으로 클래스 객체(즉 type이나 파생 클래스의 인스턴스)만 클래스로 간주합니다. 그러나, 객체는 __bases__ 어트리뷰트(베이스 클래스의 튜플이어야 합니다)를 가짐으로써 이를 재정의할 수 있습니다.

int **PyObject_IsInstance**(*PyObject* *inst, *PyObject* *cls)

Part of the 안정 ABI. *inst*가 *cls* 클래스나 *cls*의 서브 클래스의 인스턴스이면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 에러가 발생하면 -1을 반환하고 예외를 설정합니다.

*cls*가 튜플이면, *cls*의 모든 항목에 대해 검사가 수행됩니다. 적어도 하나의 검사에서 1을 반환하면 결과는 1이 되고, 그렇지 않으면 0이 됩니다.

*cls*에 __instancecheck__() 메서드가 있으면, **PEP 3119**에 설명된 대로 서브 클래스 상태를 판별하기 위해 호출됩니다. 그렇지 않으면, *inst*는 해당 클래스가 *cls*의 서브 클래스일 때 *cls*의 인스턴스입니다.

인스턴스 *inst*는 __class__ 어트리뷰트를 가짐으로써 클래스로 간주하는 것을 재정의할 수 있습니다.

객체 *cls*는 __bases__ 어트리뷰트(베이스 클래스의 튜플이어야 합니다)를 가짐으로써, 클래스로 간주하는지와 베이스 클래스가 무엇인지를 재정의할 수 있습니다.

Py_hash_t **PyObject_Hash**(*PyObject* *o)

Part of the 안정 ABI. 객체 *o*의 해시값을 계산하고 반환합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 표현식 hash(o)와 동등합니다.

버전 3.2에서 변경: 반환형은 이제 Py_hash_t입니다. 이것은 *Py_ssize_t*와 같은 크기의 부호 있는 정수입니다.

Py_hash_t **PyObject_HashNotImplemented**(*PyObject* *o)

Part of the 안정 ABI. type(o)가 해시 가능하지 않음을 나타내는 TypeError를 설정하고 -1을 반환합니다. 이 함수는 tp_hash 슬롯에 저장될 때 특수한 처방을 받아서, 인터프리터에 형이 해시 가능하지 않음을 명시적으로 알립니다.

int **PyObject_IsTrue**(*PyObject* *o)

Part of the 안정 ABI. 객체 *o*를 참으로 간주하면 1을, 그렇지 않으면 0을 반환합니다. 이것은 파이썬 표현식 not not o와 동등합니다. 실패하면 -1을 반환합니다.

int **PyObject_Not**(*PyObject* *o)

Part of the 안정 ABI. 객체 *o*를 참으로 간주하면 0을, 그렇지 않으면 1을 반환합니다. 이것은 파이썬 표현식 not o와 동등합니다. 실패하면 -1을 반환합니다.

PyObject ***PyObject_Type**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. *o*가 NULL이 아니면, 객체 *o*의 객체 형에 해당하는 형 객체를 반환합니다. 실패하면 SystemError를 발생시키고 NULL을 반환합니다. 이것은 파이썬 표현식 type(o)와 동등합니다. 이 함수는 반환 값에 대한 새로운 강한 참조를 만듭니다. 새로운 강한 참조가 필요할 때를 제외하고, *PyTypeObject** 형의 포인터를 반환하는 *Py_TYPE()* 함수 대신 이 함수를 사용할 이유가 없습니다.

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)

객체 *o*가 *type* 형이거나 *type*의 서브 형이면 0이 아닌 값을 반환하고, 그렇지 않으면 0을 반환합니다. 두 매개 변수 모두 NULL이 아니어야 합니다.

Py_ssize_t **PyObject_Size**(*PyObject* *o)

Py_ssize_t **PyObject_Length**(*PyObject* *o)

Part of the 안정 ABI. 객체 *o*의 길이를 반환합니다. 객체 *o*가 시퀀스와 매핑 프로토콜을 제공하면, 시퀀스 길이가 반환됩니다. 에러가 발생하면 -1이 반환됩니다. 이것은 파이썬 표현식 `len(o)`와 동등합니다.

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)

o 객체의 추정된 길이를 반환합니다. 먼저 실제 길이를 반환하려고 시도한 다음, `__length_hint__()`를 사용하여 추정값을 반환하고, 마지막으로 기본값을 반환합니다. 에러 시 -1을 반환합니다. 이것은 파이썬 표현식 `operator.length_hint(o, defaultvalue)`와 동등합니다.

Added in version 3.4.

PyObject ***PyObject_GetItem**(*PyObject* *o, *PyObject* *key)

반환값: 새 참조. *Part of the* 안정 ABI. 객체 *key*에 해당하는 *o*의 요소를 반환하거나 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o[key]`와 동등합니다.

int **PyObject_SetItem**(*PyObject* *o, *PyObject* *key, *PyObject* *v)

Part of the 안정 ABI. 객체 *key*를 값 *v*에 매핑합니다. 실패 시 예외를 발생시키고 -1을 반환합니다; 성공하면 0을 반환합니다. 이것은 파이썬 문장 `o[key] = v`와 동등합니다. 이 함수는 *v*에 대한 참조를 훔치지 않습니다.

int **PyObject_DelItem**(*PyObject* *o, *PyObject* *key)

Part of the 안정 ABI. 객체 *o*에서 객체 *key*에 대한 매핑을 제거합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 문장 `del o[key]`와 동등합니다.

int **PyObject_DelItemString**(*PyObject* *o, const char *key)

Part of the 안정 ABI. This is the same as *PyObject_DelItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

PyObject ***PyObject_Dir**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 이것은 파이썬 표현식 `dir(o)`와 동등하며, 객체 인자에 적합한 문자열의 (비어있을 수 있는) 리스트를 반환하거나, 에러가 있으면 NULL을 반환합니다. 인자가 NULL이면, 파이썬 `dir()`과 비슷하며, 현재 지역(locals)의 이름들을 반환합니다; 이 경우, 실행 프레임이 활성화되어 있지 않으면 NULL이 반환되지만 *PyErr_Occurred()*는 거짓을 반환합니다.

PyObject ***PyObject_GetIter**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 이것은 파이썬 표현식 `iter(o)`와 동등합니다. 객체 인자에 대한 새로운 이터레이터를 반환하거나, 객체가 이미 이터레이터이면 객체 자체를 반환합니다. 객체를 이터레이트 할 수 없으면 TypeError를 발생시키고 NULL을 반환합니다.

PyObject ***PyObject_SelfIter**(*PyObject* *obj)

반환값: 새 참조. *Part of the* 안정 ABI. This is equivalent to the Python `__iter__(self): return self` method. It is intended for *iterator* types, to be used in the *PyTypeObject.tp_iter* slot.

PyObject ***PyObject_GetAIter**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.10* 이후로. 이것은 파이썬 표현식 `aiter(o)`와 동등합니다. AsyncIterable을 받아서 그 것의 AsyncIterator를 반환합니다. 보통 새 이터레이터이지만, 인자가 AsyncIterator면 그 자신을 반환합니다. 객체를 이터레이트 할 수 없으면 TypeError를 발생시키고 NULL을 반환합니다.

Added in version 3.10.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

Part of the 안정 ABI 버전 *3.12* 이후로. Get a pointer to subclass-specific data reserved for *cls*.

The object *o* must be an instance of *cls*, and *cls* must have been created using negative *PyType_Spec.basicsize*. Python does not check this.

On error, set an exception and return `NULL`.

Added in version 3.12.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

Part of the 안정 ABI 버전 *3.12* 이후로. Return the size of the instance memory space reserved for *cls*, i.e. the size of the memory `PyObject_GetTypeData()` returns.

This may be larger than requested using `-PyType_Spec.basicsize`; it is safe to use this larger size (e.g. with `memset()`).

The type *cls* **must** have been created using negative `PyType_Spec.basicsize`. Python does not check this.

On error, set an exception and return a negative value.

Added in version 3.12.

void ***PyObject_GetItemData**(*PyObject* *o)

Get a pointer to per-item data for a class with `Py_TPFLAGS_ITEMS_AT_END`.

On error, set an exception and return `NULL`. `TypeError` is raised if *o* does not have `Py_TPFLAGS_ITEMS_AT_END` set.

Added in version 3.12.

int **PyObject_VisitManagedDict**(*PyObject* *obj, *visitproc* visit, void *arg)

Visit the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Added in version 3.13.

void **PyObject_ClearManagedDict**(*PyObject* *obj)

Clear the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Added in version 3.13.

int **PyUnstable_Object_EnableDeferredRefcount**(*PyObject* *obj)

이것은 불안정 *API*. It may change without warning in minor releases.

Enable deferred reference counting on *obj*, if supported by the runtime. In the *free-threaded* build, this allows the interpreter to avoid reference count adjustments to *obj*, which may improve multi-threaded performance. The tradeoff is that *obj* will only be deallocated by the tracing garbage collector, and not when the interpreter no longer has any references to it.

This function returns `1` if deferred reference counting is enabled on *obj*, and `0` if deferred reference counting is not supported or if the hint was ignored by the interpreter, such as when deferred reference counting is already enabled on *obj*. This function is thread-safe, and cannot fail.

This function does nothing on builds with the *GIL* enabled, which do not support deferred reference counting. This also does nothing if *obj* is not an object tracked by the garbage collector (see `gc.is_tracked()` and `PyObject_GC_IsTracked()`).

This function is intended to be used soon after *obj* is created, by the code that creates it, such as in the object's `tp_new` slot.

Added in version 3.14.

int **PyUnstable_Object_IsUniqueReferencedTemporary**(*PyObject* *obj)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Check if *obj* is a unique temporary object. Returns `1` if *obj* is known to be a unique temporary object, and `0` otherwise. This function cannot fail, but the check is conservative, and may return `0` in some cases even if *obj* is a unique temporary object.

If an object is a unique temporary, it is guaranteed that the current code has the only reference to the object. For arguments to C functions, this should be used instead of checking if the reference count is `1`. Starting with Python 3.14, the interpreter internally avoids some reference count modifications when loading objects onto the operands stack by *borrowing* references when possible, which means that a reference count of `1` by itself does not guarantee that a function argument uniquely referenced.

In the example below, `my_func` is called with a unique temporary object as its argument:

```
my_func([1, 2, 3])
```

In the example below, `my_func` is **not** called with a unique temporary object as its argument, even if its refcount is `1`:

```
my_list = [1, 2, 3]
my_func(my_list)
```

See also the function `Py_REFCNT()`.

Added in version 3.14.

int **PyUnstable_IsImmortal**(*PyObject* *obj)

> 이것은 불안정 *API*. It may change without warning in minor releases.

This function returns non-zero if *obj* is *immortal*, and zero otherwise. This function cannot fail.

> **참고**
>
> Objects that are immortal in one CPython version are not guaranteed to be immortal in another.

Added in version 3.14.

int **PyUnstable_TryIncRef**(*PyObject* *obj)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Increments the reference count of *obj* if it is not zero. Returns `1` if the object' s reference count was successfully incremented. Otherwise, this function returns `0`.

`PyUnstable_EnableTryIncRef()` must have been called earlier on *obj* or this function may spuriously return `0` in the *free threading* build.

This function is logically equivalent to the following C code, except that it behaves atomically in the *free threading* build:

```
if (Py_REFCNT(op) > 0) {
   Py_INCREF(op);
   return 1;
}
return 0;
```

This is intended as a building block for managing weak references without the overhead of a Python *weak reference object*.

Typically, correct use of this function requires support from *obj*' s deallocator (`tp_dealloc`). For example, the following sketch could be adapted to implement a "weakmap" that works like a `WeakValueDictionary` for a specific type:

```
PyMutex mutex;

PyObject *
add_entry(weakmap_key_type *key, PyObject *value)
{
    PyUnstable_EnableTryIncRef(value);
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_add_entry(weakmap, key, value);
    PyMutex_Unlock(&mutex);
    Py_RETURN_NONE;
}

PyObject *
get_value(weakmap_key_type *key)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    PyObject *result = weakmap_find(weakmap, key);
    if (PyUnstable_TryIncRef(result)) {
        // `result` is safe to use
        PyMutex_Unlock(&mutex);
        return result;
    }
    // if we get here, `result` is starting to be garbage-collected,
    // but has not been removed from the weakmap yet
    PyMutex_Unlock(&mutex);
    return NULL;
}

// tp_dealloc function for weakmap values
void
value_dealloc(PyObject *value)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_remove_value(weakmap, value);

    ...
    PyMutex_Unlock(&mutex);
}
```

Added in version 3.14.

void **PyUnstable_EnableTryIncRef**(*PyObject* *obj)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Enables subsequent uses of `PyUnstable_TryIncRef()` on *obj*. The caller must hold a *strong reference* to *obj* when calling this.

Added in version 3.14.

int **PyUnstable_Object_IsUniquelyReferenced**(*PyObject* *op)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Determine if *op* only has one reference.

On GIL-enabled builds, this function is equivalent to `Py_REFCNT(op) == 1`.

On a *free threaded* build, this checks if *op*' s *reference count* is equal to one and additionally checks if *op* is only used by this thread. `Py_REFCNT(op) == 1` is **not** thread-safe on free threaded builds; prefer this function.

The caller must hold an *attached thread state*, despite the fact that this function doesn' t call into the Python interpreter. This function cannot fail.

Added in version 3.14.

8.2 호출 프로토콜

CPython은 두 가지 호출 프로토콜을 지원합니다: *tp_call*과 벡터콜(vectorcall).

8.2.1 *tp_call* 프로토콜

`tp_call`을 설정하는 클래스의 인스턴스는 콜러블입니다. 슬롯의 서명은 다음과 같습니다:

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject *kwargs);
```

파이썬 코드의 `callable(*args, **kwargs)` 와 유사하게, 위치 인자를 위한 튜플과 키워드 인자를 위한 딕셔너리를 사용하여 호출합니다. *args*는 NULL이 아니어야 합니다 (인자가 없으면 빈 튜플을 사용하십시오). 하지만 키워드 인자가 없으면 *kwargs*는 *NULL*일 수 있습니다.

이 규칙은 *tp_call*에서만 사용되는 것이 아닙니다: `tp_new`와 `tp_init`도 인자를 이런 식으로 전달합니다.

객체를 호출하려면, `PyObject_Call()`이나 다른 호출 *API*를 사용하십시오.

8.2.2 벡터콜(Vectorcall) 프로토콜

Added in version 3.9.

벡터콜 프로토콜은 **PEP 590**에서 호출 효율을 높이기 위한 추가 프로토콜로 도입되었습니다.

경험 규칙으로, CPython은 콜러블이 지원하면 내부 호출에 대해 벡터콜을 선호합니다. 그러나 이것은 엄격한 규칙이 아닙니다. 또한, 일부 제삼자 확장은 (`PyObject_Call()`을 사용하지 않고) *tp_call*을 직접 사용합니다. 따라서, 벡터콜을 지원하는 클래스도 `tp_call`을 구현해야 합니다. 또한, 어떤 프로토콜을 사용하는지와 관계없이 콜러블은 동일하게 작동해야 합니다. 이를 위해 권장되는 방법은 `tp_call`을 `PyVectorcall_Call()`로 설정하는 것입니다. 이것이 반복을 처리합니다:

경고

벡터콜을 지원하는 클래스도 같은 의미가 있도록 *tp_call*을 **반드시** 구현해야 합니다.

버전 3.12에서 변경: The *Py_TPFLAGS_HAVE_VECTORCALL* flag is now removed from a class when the class' s `__call__()` method is reassigned. (This internally sets *tp_call* only, and thus may make it behave differently than the vectorcall function.) In earlier Python versions, vectorcall should only be used with *immutable* or static types.

*tp_call*보다 느려진다면 클래스는 벡터콜을 구현해서는 안 됩니다. 예를 들어, 피호출자가 어차피 인자를 인자 튜플과 kwargs 딕셔너리로 변환해야 하면, 벡터콜을 구현할 이유가 없습니다.

클래스는 *Py_TPFLAGS_HAVE_VECTORCALL* 플래그를 활성화하고 *tp_vectorcall_offset*을 *vectorcallfunc*가 등장하는 객체 구조체 내부의 오프셋으로 설정하여 벡터콜 프로토콜을 구현할 수 있습니다. 이것은 다음과 같은 서명을 갖는 함수를 가리키는 포인터입니다:

typedef *PyObject* *(***vectorcallfunc**)(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Part of the 안정 ABI 버전 *3.12* 이후로.

- *callable*은 호출되는 객체입니다.
- ***args*는 위치 인자와 그 뒤를 따르는 키워드 인자의 값으로 구성된 C 배열입니다.**
  인자가 없으면 *NULL*일 수 있습니다.
- ***nargsf*는 위치 인자의 수에**
  PY_VECTORCALL_ARGUMENTS_OFFSET 플래그를 더한 것입니다. *nargsf*에서 실제 위치 인자 수를 얻으려면, *PyVectorcall_NARGS()*를 사용하십시오.
- ***kwnames*는 키워드 인자의 이름을 포함하는 튜플입니다;**
  다시 말해, kwargs 딕셔너리의 키. 이 이름들은 문자열(`str`이나 서브 클래스의 인스턴스)이어야 하며 고유해야 합니다. 키워드 인자가 없으면, *kwnames*는 대신 *NULL*일 수 있습니다.

PY_VECTORCALL_ARGUMENTS_OFFSET

Part of the 안정 ABI 버전 *3.12* 이후로. 이 플래그가 벡터콜 *nargsf* 인자에 설정되면, 피호출자는 일시적으로 `args[-1]`을 변경할 수 있습니다. 즉, *args*는 할당된 벡터에서 인자 1(0이 아닙니다)을 가리킵니다. 피호출자는 반환하기 전에 `args[-1]` 값을 복원해야 합니다.

PyObject_VectorcallMethod() 의 경우, 이 플래그는 대신 `args[0]`이 변경될 수 있음을 의미합니다.

(추가 할당 없이) 저렴하게 할 수 있을 때마다, 호출자는 *PY_VECTORCALL_ARGUMENTS_OFFSET*을 사용하는 것이 좋습니다. 이렇게 하면 연결된 메서드와 같은 콜러블이 후속 호출(앞에 *self* 인자를 포함하는)을 매우 효율적으로 만들 수 있습니다.

Added in version 3.8.

벡터콜을 구현하는 객체를 호출하려면, 다른 콜러블과 마찬가지로 호출 *API* 함수를 사용하십시오. *PyObject_Vectorcall()*은 일반적으로 가장 효율적입니다.

재귀 제어

*tp_call*을 사용할 때, 피호출자는 재귀에 대해 걱정할 필요가 없습니다: CPython은 *tp_call*을 사용하여 호출한 경우 *Py_EnterRecursiveCall()* 과 *Py_LeaveRecursiveCall()* 을 사용합니다.

효율성을 위해, 벡터콜을 사용하여 호출한 경우에는 그렇지 않습니다: 피호출자는 필요하면 *Py_EnterRecursiveCall* 과 *Py_LeaveRecursiveCall* 을 사용해야 합니다.

벡터콜 지원 API

Py_ssize_t **PyVectorcall_NARGS**(size_t nargsf)

Part of the 안정 ABI 버전 *3.12* 이후로. 벡터콜 *nargsf* 인자가 주어지면, 실제 인자 수를 반환합니다. 현재 다음과 동등합니다:

```
(Py_ssize_t)(nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET)
```

그러나, 향후 확장을 위해 `PyVectorcall_NARGS` 함수를 사용해야 합니다.

Added in version 3.8.

vectorcallfunc **PyVectorcall_Function**(*PyObject* *op)

*op*가 벡터콜 프로토콜을 지원하지 않으면 (형이 지원하지 않거나 인스턴스가 지원하지 않기 때문에), *NULL*을 반환합니다. 그렇지 않으면, *op*에 저장된 벡터콜 함수 포인터를 반환합니다. 이 함수는 예외를 발생시키지 않습니다.

이것은 *op*가 벡터콜을 지원하는지를 확인하는 데 주로 유용하며, `PyVectorcall_Function(op) != NULL`을 확인하여 수행 할 수 있습니다.

Added in version 3.9.

PyObject ***PyVectorcall_Call**(*PyObject* *callable, *PyObject* *tuple, *PyObject* *dict)

Part of the 안정 ABI 버전 *3.12* 이후로. 튜플과 딕셔너리에 각각 주어진 위치와 키워드 인자로 *callable*의 *vectorcallfunc*를 호출합니다.

이것은 *tp_call* 슬롯에 넣거나 `tp_call` 구현에 사용하기 위한 특수 함수입니다. *Py_TPFLAGS_HAVE_VECTORCALL* 플래그를 확인하지 않고 `tp_call`로 폴백하지 않습니다.

Added in version 3.8.

8.2.3 객체 호출 API

파이썬 객체를 호출하기 위해 다양한 함수를 사용할 수 있습니다. 각각은 인자를 호출된 객체가 지원하는 규칙으로 변환합니다 - *tp_call* 또는 벡터콜. 가능한 한 적은 변환을 수행하려면, 사용 가능한 데이터 형식에 가장 적합한 것을 선택하십시오.

다음 표는 사용 가능한 함수를 요약한 것입니다; 자세한 내용은 개별 설명서를 참조하십시오.

함수	콜러블	args	kwargs
PyObject_Call()	`PyObject *`	튜플	딕셔너리/`NULL`
PyObject_CallNoArgs()	`PyObject *`	—	—
PyObject_CallOneArg()	`PyObject *`	1 객체	—
PyObject_CallObject()	`PyObject *`	튜플/`NULL`	—
PyObject_CallFunction()	`PyObject *`	포맷 (format)	—
PyObject_CallMethod()	obj + `char*`	포맷 (format)	—
PyObject_CallFunctionObjArgs()	`PyObject *`	가변 (variadic)	—
PyObject_CallMethodObjArgs()	obj + name	가변 (variadic)	—
PyObject_CallMethodNoArgs()	obj + name	—	—
PyObject_CallMethodOneArg()	obj + name	1 객체	—
PyObject_Vectorcall()	`PyObject *`	벡터콜	벡터콜
PyObject_VectorcallDict()	`PyObject *`	벡터콜	딕셔너리/`NULL`
PyObject_VectorcallMethod()	arg + name	벡터콜	벡터콜

PyObject ***PyObject_Call**(*PyObject* *callable, *PyObject* *args, *PyObject* *kwargs)

반환값: 새 참조. *Part of the* 안정 ABI. 튜플 *args*로 주어진 인자와 딕셔너리 *kwargs*로 주어진 이름있는 인자로 콜러블 파이썬 객체 *callable*을 호출합니다.

*args*는 *NULL*이 아니어야 합니다 | 인자가 필요 없으면 빈 튜플을 사용하십시오. 이름있는 인자가 필요하지 않으면, *kwargs*는 *NULL*일 수 있습니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

이것은 다음 파이썬 표현식과 동등합니다: `callable(*args, **kwargs)`.

PyObject ***PyObject_CallNoArgs**(*PyObject* *callable)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.10* 이후로. 인자 없이 콜러블 파이썬 객체 *callable*을 호출합니다. 인자 없이 콜러블 파이썬 객체를 호출하는 가장 효율적인 방법입니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

Added in version 3.9.

PyObject ***PyObject_CallOneArg**(*PyObject* *callable, *PyObject* *arg)

반환값: 새 참조. 정확히 1개의 위치 인자 *arg*로 키워드 인자 없이 콜러블 파이썬 객체 *callable*을 호출합니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

Added in version 3.9.

PyObject ***PyObject_CallObject**(*PyObject* *callable, *PyObject* *args)

반환값: 새 참조. *Part of the* 안정 ABI. 튜플 *args*에 의해 주어진 인자로 콜러블 파이썬 객체 *callable*을 호출합니다. 인자가 필요하지 않으면 *args*는 *NULL*일 수 있습니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

이것은 다음 파이썬 표현식과 동등합니다: `callable(*args)`.

PyObject ***PyObject_CallFunction**(*PyObject* *callable, const char *format, ...)

반환값: 새 참조. *Part of the* 안정 ABI. 가변 개수의 C 인자로 콜러블 파이썬 객체 *callable*을 호출합니다. C 인자는 *Py_BuildValue()* 스타일 포맷 문자열을 사용하여 기술됩니다. format은 *NULL*일 수 있으며, 인자가 제공되지 않음을 나타냅니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

이것은 다음 파이썬 표현식과 동등합니다: `callable(*args)`.

*PyObject** 인자만 전달한다면 *PyObject_CallFunctionObjArgs()* 가 더 빠른 대안임에 유의하십시오.

버전 3.4에서 변경: *format*의 형이 `char *`에서 변경되었습니다.

PyObject ***PyObject_CallMethod**(*PyObject* *obj, const char *name, const char *format, ...)

반환값: 새 참조. *Part of the* 안정 ABI. 가변 개수의 C 인자를 사용하여 객체 *obj*의 *name*이라는 이름의 메서드를 호출합니다. C 인자는 튜플을 생성해야 하는 *Py_BuildValue()* 포맷 문자열로 기술됩니다.

format은 *NULL*일 수 있으며, 인자가 제공되지 않음을 나타냅니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

이것은 다음 파이썬 표현식과 동등합니다: `obj.name(arg1, arg2, ...)`.

*PyObject** 인자만 전달한다면 *PyObject_CallMethodObjArgs()* 가 더 빠른 대안임에 유의하십시오.

버전 3.4에서 변경: *name*과 *format*의 형이 `char *`에서 변경되었습니다.

PyObject ***PyObject_CallFunctionObjArgs**(*PyObject* *callable, ...)

반환값: 새 참조. *Part of the* 안정 ABI. 가변 개수의 *PyObject** 인자로 콜러블 파이썬 객체 *callable*을 호출합니다. 인자는 *NULL*로 끝나는 가변 개수의 매개 변수로 제공됩니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

이것은 다음 파이썬 표현식과 동등합니다: `callable(arg1, arg2, ...)`.

PyObject ***PyObject_CallMethodObjArgs**(*PyObject* *obj, *PyObject* *name, ...)

반환값: 새 참조. *Part of the* 안정 ABI. 파이썬 객체 *obj*의 메서드를 호출합니다. 여기서 메서드 이름은 *name*에서 파이썬 문자열 객체로 제공됩니다. 가변 개수의 *PyObject** 인자로 호출됩니다. 인자는 *NULL*로 끝나는 가변 개수의 매개 변수로 제공됩니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

PyObject ***PyObject_CallMethodNoArgs**(*PyObject* *obj, *PyObject* *name)

인자 없이 파이썬 객체 *obj*의 메서드를 호출합니다. 여기서 메서드 이름은 *name*에서 파이썬 문자열 객체로 제공됩니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

Added in version 3.9.

PyObject ***PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

단일 위치 인자 *arg*로 파이썬 객체 *obj*의 메서드를 호출합니다. 여기서 메서드 이름은 *name*에서 파이썬 문자열 객체로 제공됩니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

Added in version 3.9.

PyObject ***PyObject_Vectorcall**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Part of the 안정 ABI 버전 *3.12* 이후로. 콜러블 파이썬 객체 *callable*을 호출합니다. 인자는 `vectorcallfunc` 와 같습니다. *callable*이 벡터콜을 지원하면, *callable*에 저장된 벡터콜 함수를 직접 호출합니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

Added in version 3.9.

PyObject ***PyObject_VectorcallDict**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwdict)

위치 인자가 벡터콜 프로토콜과 정확히 일치하지만 딕셔너리 *kwdict*로 전달된 키워드 인자로 *callable*을 호출합니다. *args* 배열은 위치 인자만 포함합니다.

내부적으로 사용되는 프로토콜과 관계없이, 인자를 변환해야 합니다. 따라서, 이 함수는 호출자에게 이미 키워드 인자로 사용할 준비가 된 딕셔너리가 있지만, 위치 인자에 대한 튜플이 없을 때만 사용해야 합니다.

Added in version 3.9.

PyObject ***PyObject_VectorcallMethod**(*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Part of the 안정 ABI 버전 *3.12* 이후로. 벡터콜 호출 규칙을 사용하여 메서드를 호출합니다. 메서드 이름은 파이썬 문자열 *name*으로 제공됩니다. 메서드가 호출되는 객체는 *args*[*0*]이며, *args*[*1*]에서 시작하는 *args* 배열은 호출의 인자를 나타냅니다. 하나 이상의 위치 인자가 있어야 합니다. *nargsf*는 *args*[*0*]을 포함한 위치 인자의 수이며, `args[0]` 값이 일시적으로 변경될 수 있으면 `PY_VECTORCALL_ARGUMENTS_OFFSET`이 더해집니다. 키워드 인자는 `PyObject_Vectorcall()`에서처럼 전달될 수 있습니다.

객체에 `Py_TPFLAGS_METHOD_DESCRIPTOR` 기능이 있으면, 전체 *args* 벡터를 인자로 연결되지 않은 메서드 객체를 호출합니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 *NULL*을 반환합니다.

Added in version 3.9.

8.2.4 호출 지원 API

int **PyCallable_Check**(*PyObject* *o)

Part of the 안정 ABI. 객체 *o*가 콜러블 인지 판별합니다. 객체가 콜러블 이면 `1`을, 그렇지 않으면 `0`을 반환합니다. 이 함수는 항상 성공합니다.

8.3 숫자 프로토콜

int **PyNumber_Check**(*PyObject* *o)

Part of the 안정 ABI. 객체 *o*가 숫자 프로토콜을 제공하면 `1`을 반환하고, 그렇지 않으면 거짓을 반환합니다. 이 함수는 항상 성공합니다.

버전 3.8에서 변경: *o*가 인덱스 정수면 `1`을 반환합니다.

PyObject ***PyNumber_Add**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*과 *o2*를 더한 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 + o2`와 동등합니다.

PyObject ***PyNumber_Subtract**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*에서 *o2*를 뺀 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 - o2`와 동등합니다.

PyObject ***PyNumber_Multiply**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*과 *o2*를 곱한 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 * o2`와 동등합니다.

PyObject ***PyNumber_MatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. *o1*과 *o2*를 행렬 곱셈한 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 @ o2`와 동등합니다.

Added in version 3.5.

PyObject ***PyNumber_FloorDivide**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*을 *o2*로 정수 나눗셈한 몫을, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 // o2`와 동등합니다.

PyObject ***PyNumber_TrueDivide**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*을 *o2*로 나눈 수학적 값의 적절한 근삿값이나, 실패 시 `NULL`을 반환합니다. 반환 값은 "근사치" 인데, 이진 부동 소수점 수가 근사치이기 때문입니다; 이진수로 모든 실수를 표현할 수는 없습니다. 이 함수는 두 개의 정수를 전달할 때 부동 소수점 수를 반환할 수 있습니다.

PyObject ***PyNumber_Remainder**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*을 *o2*로 나눈 나머지나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 % o2`와 동등합니다.

PyObject ***PyNumber_Divmod**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 내장 함수 `divmod()`를 참조하십시오. 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `divmod(o1, o2)`와 동등합니다.

PyObject ***PyNumber_Power**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

반환값: 새 참조. *Part of the* 안정 ABI. 내장 함수 `pow()`를 참조하십시오. 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `pow(o1, o2, o3)`와 동등합니다, 여기서 *o3*는 선택적입니다. *o3*를 무시하려면, 그 자리에 *Py_None*을 전달하십시오 (*o3*에 `NULL`을 전달하면 잘못된 메모리 액세스가 발생합니다).

PyObject ***PyNumber_Negative**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o*의 음의 값(negation)을, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `-o`와 동등합니다.

PyObject ***PyNumber_Positive**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o*를, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `+o`와 동등합니다.

PyObject ***PyNumber_Absolute**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. *o*의 절댓값이나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `abs(o)`와 동등합니다.

PyObject ***PyNumber_Invert**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o*의 비트 반전(bitwise negation)을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `~o`와 동등합니다.

PyObject ***PyNumber_Lshift**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*을 *o2*만큼 왼쪽으로 시프트 한 결과를, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o1 << o2`와 동등합니다.

PyObject ***PyNumber_Rshift**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*을 *o2*만큼 오른쪽으로 시프트 한 결과를, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o1 >> o2`와 동등합니다.

PyObject ***PyNumber_And**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*과 *o2*의 "비트별 논리곱(bitwise and)"을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o1 & o2`와 동등합니다.

PyObject ***PyNumber_Xor**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*과 *o2*의 "비트별 배타적 논리합(bitwise exclusive or)"을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o1 ^ o2`와 동등합니다.

PyObject ***PyNumber_Or**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*과 *o2*의 "비트별 논리합(bitwise or)"을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o1 | o2`와 동등합니다.

PyObject ***PyNumber_InPlaceAdd**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*과 *o2*를 더한 결과나, 실패 시 NULL을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 += o2`와 동등합니다.

PyObject ***PyNumber_InPlaceSubtract**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*에서 *o2*를 뺀 결과나, 실패 시 NULL을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 -= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceMultiply**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*과 *o2*를 곱한 결과나, 실패 시 NULL을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 *= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceMatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. *o1*과 *o2*를 행렬 곱셈한 결과나, 실패 시 NULL을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 @= o2`와 동등합니다.

Added in version 3.5.

PyObject ***PyNumber_InPlaceFloorDivide**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*을 *o2*로 나눈 수학적 플로어(floor)나, 실패 시 NULL을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 //= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceTrueDivide**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*을 *o2*로 나눈 수학적 값의 적절한 근삿값이나, 실패 시 NULL을 반환합니다. 반환 값은 "근사치" 인데, 이진 부동 소수점 수가 근사치이기 때문입니다; 이진수로 모든 실수를 표현할 수는 없습니다. 이 함수는 두 개의 정수를 전달할 때 부동 소수점 수를 반환할 수 있습니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 /= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceRemainder**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. *o1*을 *o2*로 나눈 나머지나, 실패 시 NULL을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 %= o2`와 동등합니다.

PyObject ***PyNumber_InPlacePower**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

반환값: 새 참조. *Part of the* 안정 ABI. 내장 함수 `pow()`를 참조하십시오. 실패하면 `NULL`을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 o3가 *Py_None*일 때 파이썬 문장 `o1 **= o2`와, 그렇지 않으면 `pow(o1, o2, o3)`의 제자리 변형과 동등합니다. *o3*를 무시하려면, 그 자리에 *Py_None*을 전달하십시오 (*o3*에 `NULL`을 전달하면 잘못된 메모리 액세스가 발생합니다).

PyObject ***PyNumber_InPlaceLshift**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*을 *o2*만큼 왼쪽으로 시프트 한 결과를, 실패 시 `NULL`을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 <<= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceRshift**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*을 *o2*만큼 오른쪽으로 시프트 한 결과를, 실패 시 `NULL`을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 >>= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceAnd**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*과 *o2*의 "비트별 논리곱(bitwise and)"을, 실패 시 `NULL`을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 &= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceXor**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*과 *o2*의 "비트별 배타적 논리합(bitwise exclusive or)"을, 실패 시 `NULL`을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 ^= o2`와 동등합니다.

PyObject ***PyNumber_InPlaceOr**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*과 *o2*의 "비트별 논리합(bitwise or)"을, 실패 시 `NULL`을 반환합니다. 이 연산은 *o1*이 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 문장 `o1 |= o2`와 동등합니다.

PyObject ***PyNumber_Long**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 정수 객체로 변환된 *o*를, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `int(o)`와 동등합니다.

PyObject ***PyNumber_Float**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 float 객체로 변환된 *o*를, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `float(o)`와 동등합니다.

PyObject ***PyNumber_Index**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 파이썬 int로 변환된 *o*를, 실패 시 `NULL`을 반환합니다. 실패 시 `TypeError` 예외가 발생합니다.

버전 3.10에서 변경: The result always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`.

PyObject ***PyNumber_ToBase**(*PyObject* *n, int base)

반환값: 새 참조. *Part of the* 안정 ABI. 정수 *n*을 진수 *base*를 사용해서 변환한 문자열을 반환합니다. *base* 인자는 2, 8, 10 또는 16중 하나여야 합니다. 진수 2, 8 또는 16의 경우, 반환된 문자열은 `'0b'`, `'0o'` 또는 `'0x'`의 진수 표시자가 각각 앞에 붙습니다. *n*이 파이썬 int가 아니면, 먼저 *PyNumber_Index()*로 변환됩니다.

Py_ssize_t **PyNumber_AsSsize_t**(*PyObject* *o, *PyObject* *exc)

Part of the 안정 ABI. *o*가 정수로 해석될 수 있으면, *o*를 *Py_ssize_t* 값으로 변환하여 반환합니다. 호출이 실패하면, 예외가 발생하고 `-1`이 반환됩니다.

*o*가 파이썬 int로 변환될 수 있지만 *Py_ssize_t* 값으로 변환하려는 시도가 `OverflowError`를 발생시키면, *exc* 인자는 발생할 예외의 형(일반적으로 `IndexError`나 `OverflowError`)입니다. *exc*가 `NULL`이면, 예외가 지워지고 값은 음의 정수는 `PY_SSIZE_T_MIN`으로, 양의 정수는 `PY_SSIZE_T_MAX`로 잘립니다.

int **PyIndex_Check**(*PyObject* *o)

Part of the 안정 ABI 버전 *3.8* 이후로. *o*가 인덱스 정수(`tp_as_number` 구조의 `nb_index` 슬롯이 채워져 있습니다)면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. 이 함수는 항상 성공합니다.

8.4 시퀀스 프로토콜

int **PySequence_Check**(*PyObject* *o)

Part of the 안정 ABI. 객체가 시퀀스 프로토콜을 제공하면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. `__getitem__()` 메서드가 있는 파이썬 클래스의 경우 `dict` 서브 클래스가 아닌 한 `1`을 반환하는 것에 유의하십시오. 일반적으로 클래스가 어떤 형의 키를 지원하는지 판단할 수 없기 때문입니다. 이 함수는 항상 성공합니다.

Py_ssize_t **PySequence_Size**(*PyObject* *o)

Py_ssize_t **PySequence_Length**(*PyObject* *o)

Part of the 안정 ABI. 성공 시 시퀀스 *o*의 객체 수를 반환하고, 실패하면 `-1`을 반환합니다. 이것은 파이썬 표현식 `len(o)`와 동등합니다.

PyObject ***PySequence_Concat**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*와 *o2*의 이어붙이기를 반환하고, 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 + o2`와 동등합니다.

PyObject ***PySequence_Repeat**(*PyObject* *o, *Py_ssize_t* count)

반환값: 새 참조. *Part of the* 안정 ABI. 시퀀스 객체 *o*를 *count* 번 반복한 결과를 반환하거나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o * count`와 동등합니다.

PyObject ***PySequence_InPlaceConcat**(*PyObject* *o1, *PyObject* *o2)

반환값: 새 참조. *Part of the* 안정 ABI. 성공 시 *o1*와 *o2*의 이어붙이기를 반환하고, 실패하면 `NULL`을 반환합니다. 이 연산은 *o1*가 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 표현식 `o1 += o2`와 동등합니다.

PyObject ***PySequence_InPlaceRepeat**(*PyObject* *o, *Py_ssize_t* count)

반환값: 새 참조. *Part of the* 안정 ABI. 시퀀스 객체 *o*를 *count* 번 반복한 결과를 반환하거나, 실패 시 `NULL`을 반환합니다. 이 연산은 *o*가 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 표현식 `o *= count`와 동등합니다.

PyObject ***PySequence_GetItem**(*PyObject* *o, *Py_ssize_t* i)

반환값: 새 참조. *Part of the* 안정 ABI. *o*의 *i* 번째 요소를 반환하거나, 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o[i]`와 동등합니다.

PyObject ***PySequence_GetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

반환값: 새 참조. *Part of the* 안정 ABI. 시퀀스 객체 *o*의 *i1*와 *i2* 사이의 슬라이스를 반환하거나, 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o[i1:i2]`와 동등합니다.

int **PySequence_SetItem**(*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

Part of the 안정 ABI. 객체 *v*를 *o*의 *i* 번째 요소에 대입합니다. 실패하면 예외를 발생시키고 `-1`을 반환합니다; 성공하면 `0`을 반환합니다. 이것은 파이썬 문장 `o[i] = v`와 동등합니다. 이 함수는 *v*에 대한 참조를 훔치지 않습니다.

*v*가 `NULL`이면, 요소가 삭제되지만, 이 기능은 *PySequence_DelItem()* 사용을 위해 폐지되었습니다.

int **PySequence_DelItem**(*PyObject* *o, *Py_ssize_t* i)

Part of the 안정 ABI. *o* 객체의 *i* 번째 요소를 삭제합니다. 실패하면 `-1`을 반환합니다. 이것은 파이썬 문장 `del o[i]`와 동등합니다.

int **PySequence_SetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2, *PyObject* *v)

Part of the 안정 ABI. 시퀀스 객체 *v*를 시퀀스 객체 *o*의 *i1*에서 *i2* 사이의 슬라이스에 대입합니다. 이것은 파이썬 문장 `o[i1:i2] = v`와 동등합니다.

int **PySequence_DelSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Part of the 안정 ABI. 시퀀스 객체 *o*의 *i1*에서 *i2* 사이의 슬라이스를 삭제합니다. 실패하면 `-1`을 반환합니다. 이것은 파이썬 문장 `del o[i1:i2]`와 동등합니다.

Py_ssize_t **PySequence_Count**(*PyObject* *o, *PyObject* *value)

Part of the 안정 ABI. *o*에 있는 *value*의 수를 반환합니다. 즉, `o[key] == value`를 만족하는 key의 수를 반환합니다. 실패하면 `-1`을 반환합니다. 이것은 파이썬 표현식 `o.count(value)`와 동등합니다.

int **PySequence_Contains**(*PyObject* *o, *PyObject* *value)

Part of the 안정 ABI. *o*에 *value*가 있는지 확인합니다. *o*의 항목 중 하나가 *value*와 같으면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. 에러 시 `-1`을 반환합니다. 이는 파이썬 표현식 `value in o`와 동등합니다.

int **PySequence_In**(*PyObject* *o, *PyObject* *value)

Part of the 안정 ABI. Alias for *PySequence_Contains()*.

버전 3.14부터 폐지됨: The function is *soft deprecated* and should no longer be used to write new code.

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

Part of the 안정 ABI. `o[i] == value`을 만족하는 첫 번째 인덱스 *i*를 반환합니다. 에러 시 `-1`을 반환합니다. 이것은 파이썬 표현식 `o.index(value)`와 동등합니다.

PyObject ***PySequence_List**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 시퀀스나 이터러블 *o*와 같은 내용을 가진 리스트 객체를 반환하거나, 실패하면 `NULL`을 반환합니다. 반환된 리스트는 새로운 것으로 보장됩니다. 이것은 파이썬 표현식 `list(o)`와 동등합니다.

PyObject ***PySequence_Tuple**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 시퀀스나 이터러블 *o*와 같은 내용을 가진 튜플 객체를 반환하거나, 실패하면 `NULL`을 반환합니다. *o*가 튜플이면, 새로운 참조가 반환되고, 그렇지 않으면 튜플이 적절한 내용으로 만들어집니다. 이것은 파이썬 표현식 `tuple(o)`와 동등합니다.

PyObject ***PySequence_Fast**(*PyObject* *o, const char *m)

반환값: 새 참조. *Part of the* 안정 ABI. 시퀀스나 이터러블 *o*를 다른 `PySequence_Fast*` 계열 함수에서 사용할 수 있는 객체로 반환합니다. 객체가 시퀀스나 이터러블이 아니면 *m*을 메시지 텍스트로 사용하여 `TypeError`를 발생시킵니다. 실패 시 `NULL`을 반환합니다.

`PySequence_Fast*` 함수는 *o*가 *PyTupleObject*나 *PyListObject*라고 가정하고 *o*의 데이터 필드에 직접 액세스하기 때문에 이렇게 이름 붙였습니다.

CPython 구현 세부 사항으로, *o*가 이미 시퀀스나 리스트면, 반환됩니다.

Py_ssize_t **PySequence_Fast_GET_SIZE**(*PyObject* *o)

*o*의 길이를 반환하는데, *o*가 *PySequence_Fast()*에 의해 반환되었고, *o*가 `NULL`이 아니라고 가정합니다. 크기는 *o*에 대해 *PySequence_Size()*를 호출하여 얻을 수도 있지만, *PySequence_Fast_GET_SIZE()*는 *o*가 리스트나 튜플이라고 가정할 수 있으므로 더 빠릅니다.

PyObject ***PySequence_Fast_GET_ITEM**(*PyObject* *o, *Py_ssize_t* i)

반환값: 빌린 참조. *o*의 *i* 번째 요소를 반환하는데, *o*가 *PySequence_Fast()*에 의해 반환되었고, *o*가 `NULL`이 아니며, *i*가 경계 내에 있다고 가정합니다.

PyObject ****PySequence_Fast_ITEMS**(*PyObject* *o)

PyObject 포인터의 하부 배열을 반환합니다. *o*가 *PySequence_Fast()*에 의해 반환되었고, *o*가 `NULL`이 아니라고 가정합니다.

리스트의 크기가 변경되면, 재할당이 항목 배열을 재배치할 수 있음에 유의하십시오. 따라서, 시퀀스가 변경될 수 없는 문맥에서만 하부 배열 포인터를 사용하십시오.

PyObject ***PySequence_ITEM**(*PyObject* *o, *Py_ssize_t* i)

반환값: 새 참조. *o*의 *i* 번째 요소를 반환하거나, 실패하면 `NULL`을 반환합니다. *PySequence_GetItem()*의 빠른 형식이지만, *o*에 대해 *PySequence_Check()*가 참인지 검사하지 않고, 음수 인덱스를 조정하지 않습니다.

8.5 매핑 프로토콜

PyObject_GetItem(), *PyObject_SetItem()* 및 *PyObject_DelItem()*도 참조하십시오.

int **PyMapping_Check**(*PyObject* *o)

Part of the 안정 ABI. 객체가 매핑 프로토콜을 제공하거나 슬라이싱을 지원하면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. `__getitem__()` 메서드가 있는 파이썬 클래스의 경우 `1`을 반환한다는 점에 유의하십시오; 일반적으로 클래스가 지원하는 키 형을 판단할 수 없기 때문입니다. 이 함수는 항상 성공합니다.

Py_ssize_t **PyMapping_Size**(*PyObject* *o)

Py_ssize_t **PyMapping_Length**(*PyObject* *o)

Part of the 안정 ABI. 성공 시 객체 *o*의 키 수를 반환하고, 실패하면 `-1`을 반환합니다. 이는 파이썬 표현식 `len(o)`와 동등합니다.

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

반환값: 새 참조. *Part of the* 안정 ABI. This is the same as *PyObject_GetItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

int **PyMapping_GetOptionalItem**(*PyObject* *obj, *PyObject* *key, *PyObject* **result)

Part of the 안정 ABI 버전 *3.13* 이후로. Variant of *PyObject_GetItem()* which doesn' t raise `KeyError` if the key is not found.

If the key is found, return `1` and set **result* to a new *strong reference* to the corresponding value. If the key is not found, return `0` and set **result* to `NULL`; the `KeyError` is silenced. If an error other than `KeyError` is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int **PyMapping_GetOptionalItemString**(*PyObject* *obj, const char *key, *PyObject* **result)

Part of the 안정 ABI 버전 *3.13* 이후로. This is the same as *PyMapping_GetOptionalItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Added in version 3.13.

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

Part of the 안정 ABI. This is the same as *PyObject_SetItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

int **PyMapping_DelItem**(*PyObject* *o, *PyObject* *key)

이것은 *PyObject_DelItem()*의 별칭입니다.

int **PyMapping_DelItemString**(*PyObject* *o, const char *key)

This is the same as *PyObject_DelItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

int **PyMapping_HasKeyWithError**(*PyObject* *o, *PyObject* *key)

Part of the 안정 ABI 버전 *3.13* 이후로. 매핑 객체에 *key* 키가 있으면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. 이는 파이썬 표현식 `key in o`와 동등합니다. 실패하면 `-1`을 반환합니다.

Added in version 3.13.

int **PyMapping_HasKeyStringWithError**(*PyObject* *o, const char *key)

Part of the 안정 ABI 버전 *3.13* 이후로. This is the same as *PyMapping_HasKeyWithError()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Added in version 3.13.

int **PyMapping_HasKey**(*PyObject* *o, *PyObject* *key)

Part of the 안정 ABI. 매핑 객체에 *key* 키가 있으면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. 이는 파이썬 표현식 `key in o`와 동등합니다. 이 함수는 항상 성공합니다.

> **참고**
>
> 이 함수가 `__getitem__()` 메서드를 호출하는 동안 발생하는 예외는 조용히 억제됩니다. 적절한 에러 처리가 필요하면 대신 *PyMapping_HasKeyWithError()*, *PyMapping_GetOptionalItem()* 또는 *PyObject_GetItem()*을 사용하십시오.

int **PyMapping_HasKeyString**(*PyObject* *o, const char *key)

Part of the 안정 ABI. This is the same as *PyMapping_HasKey()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

> **참고**
>
> 이 함수가 `__getitem__()` 메서드를 호출하거나 임시 `str` 객체를 만드는 동안 발생하는 예외는 조용히 억제됩니다. 적절한 에러 처리가 필요하면 대신 *PyMapping_HasKeyStringWithError()*, *PyMapping_GetOptionalItemString()* 또는 *PyMapping_GetItemString()*을 사용하십시오.

PyObject ***PyMapping_Keys**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공하면, 객체 *o*의 키 리스트를 반환합니다. 실패하면, `NULL`을 반환합니다.

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

PyObject ***PyMapping_Values**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공하면, 객체 *o*의 값 리스트를 반환합니다. 실패하면, `NULL`을 반환합니다.

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

PyObject ***PyMapping_Items**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 성공하면, 객체 *o*에 있는 항목 리스트를 반환합니다. 여기서 각 항목은 키-값 쌍을 포함하는 튜플입니다. 실패하면, `NULL`을 반환합니다.

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

8.6 이터레이터 프로토콜

특히 이터레이터를 사용하기 위한 두 함수가 있습니다.

int **PyIter_Check**(*PyObject* *o)

Part of the 안정 ABI 버전 *3.8* 이후로. Return non-zero if the object *o* can be safely passed to *PyIter_NextItem()* and `0` otherwise. This function always succeeds.

int **PyAIter_Check**(*PyObject* *o)

Part of the 안정 ABI 버전 *3.10* 이후로. 객체 *o*가 `AsyncIterator` 프로토콜을 지원하면 `0`이 아닌 값을, 그렇지 않으면 `0`을 돌려줍니다. 이 함수는 항상 성공합니다.

Added in version 3.10.

int **PyIter_NextItem**(*PyObject* *iter, *PyObject* **item)

Part of the 안정 ABI 버전 *3.14* 이후로. Return `1` and set *item* to a *strong reference* of the next value of the iterator *iter* on success. Return `0` and set *item* to `NULL` if there are no remaining values. Return `-1`, set *item* to `NULL` and set an exception on error.

Added in version 3.14.

PyObject ***PyIter_Next**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. This is an older version of `PyIter_NextItem()`, which is retained for backwards compatibility. Prefer *PyIter_NextItem()*.

이터레이터 *o*에서 다음 값을 반환합니다. 객체는 *PyIter_Check()*에 따르는 이터레이터 여야 합니다 (이것을 확인하는 것은 호출자 책임입니다). 남은 값이 없으면, 예외가 설정되지 않은 상태로 NULL을 반환합니다. 항목을 꺼내는 동안 에러가 발생하면, NULL을 반환하고 예외를 전달합니다.

type **PySendResult**

The enum value used to represent different results of *PyIter_Send()*.

Added in version 3.10.

PySendResult **PyIter_Send**(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

Part of the 안정 ABI 버전 *3.10* 이후로. Sends the *arg* value into the iterator *iter*. Returns:

- PYGEN_RETURN if iterator returns. Return value is returned via *presult*.
- PYGEN_NEXT if iterator yields. Yielded value is returned via *presult*.
- PYGEN_ERROR if iterator has raised and exception. *presult* is set to NULL.

Added in version 3.10.

8.7 버퍼 프로토콜

파이썬에서 사용할 수 있는 어떤 객체는 하부 메모리 배열 또는 버퍼에 대한 액세스를 감쌉니다. 이러한 객체에는 내장 bytes 와 bytearray, 그리고 array.array와 같은 일부 확장형이 포함됩니다. 제삼자 라이브러리도 이미지 처리나 수치 해석과 같은 특수한 용도로 자체 형을 정의할 수 있습니다.

이러한 형은 각각 고유의 의미가 있지만, (아마도) 큰 메모리 버퍼에 의해 뒷받침되는 공통된 특징을 공유합니다. 어떤 상황에서는 중간 복사 없이 직접 버퍼에 액세스하는 것이 바람직합니다.

Python provides such a facility at the C and Python level in the form of the *buffer protocol*. This protocol has two sides:

- on the producer side, a type can export a "buffer interface" which allows objects of that type to expose information about their underlying buffer. This interface is described in the section 버퍼 객체 구조체; for Python see python-buffer-protocol.
- on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter). For Python see memoryview.

bytes 와 bytearray와 같은 간단한 객체는 하부 버퍼를 바이트 지향 형식으로 노출합니다. 다른 형태도 가능합니다; 예를 들어, array.array에 의해 노출되는 요소는 멀티 바이트 값이 될 수 있습니다.

버퍼 인터페이스의 소비자 예는 파일 객체의 write() 메서드입니다: 버퍼 인터페이스를 통해 일련의 바이트를 내보낼 수 있는 모든 객체는 파일에 기록될 수 있습니다. write()가 전달된 객체의 내부 내용에 대한 읽기 전용 액세스만 필요하지만, readinto()와 같은 다른 메서드는 인자의 내용에 쓰기 액세스가 필요합니다. 버퍼 인터페이스는 객체가 읽기-쓰기와 읽기 전용 버퍼를 선택적으로 허용하거나 거부할 수 있도록 합니다.

버퍼 인터페이스의 소비자가 대상 객체에 대해 버퍼를 얻는 방법에는 두 가지가 있습니다:

- 올바른 매개 변수로 *PyObject_GetBuffer()*를 호출합니다;
- y*, w* 또는 s* 형식 코드 중 하나를 사용하여 *PyArg_ParseTuple()* (또는 그 형제 중 하나)을 호출합니다.

두 경우 모두, 버퍼가 더는 필요하지 않으면 *PyBuffer_Release()*를 호출해야 합니다. 그렇게 하지 않으면 자원 누수와 같은 다양한 문제가 발생할 수 있습니다.

Added in version 3.12: The buffer protocol is now accessible in Python, see python-buffer-protocol and memoryview.

8.7.1 버퍼 구조체

버퍼 구조체 (또는 단순히 "버퍼")는 다른 객체의 바이너리 데이터를 파이썬 프로그래머에게 노출하는 방법으로 유용합니다. 또한, 복사 없는 (zero-copy) 슬라이싱 메커니즘으로 사용할 수 있습니다. 메모리 블록을 참조하는 능력을 사용해서, 임의의 데이터를 파이썬 프로그래머에게 아주 쉽게 노출할 수 있습니다. 메모리는 C 확장의 큰 상수 배열일 수 있으며, 운영 체제 라이브러리로 전달되기 전에 조작하기 위한 원시

메모리 블록일 수도 있고, 네이티브 인 메모리(in-memory) 형식으로 구조화된 데이터를 전달하는 데 사용될 수도 있습니다.

파이썬 인터프리터가 노출하는 대부분의 데이터형과 달리, 버퍼는 *PyObject* 포인터가 아니라 단순한 C 구조체입니다. 이를 통해 매우 간단하게 만들고 복사할 수 있습니다. 버퍼를 감싸는 일반 래퍼가 필요할 때는, 메모리 뷰 객체를 만들 수 있습니다.

제공하는(exporting) 객체를 작성하는 간단한 지침은 버퍼 객체 구조체를 참조하십시오. 버퍼를 얻으려면, *PyObject_GetBuffer()*를 참조하십시오.

type **Py_buffer**

Part of the 안정 ABI (*including all members*) 버전 *3.11* 이후로.

void ***buf**

버퍼 필드에 의해 기술된 논리적 구조의 시작을 가리키는 포인터. 이것은 제공자(exporter)의 하부 물리적 메모리 블록 내의 모든 위치일 수 있습니다. 예를 들어, 음의 *strides*를 사용하면 값이 메모리 블록의 끝을 가리킬 수 있습니다.

연속 배열의 경우, 값은 메모리 블록의 시작을 가리킵니다.

PyObject ***obj**

제공하는(exporting) 객체에 대한 새 참조. 참조는 소비자가 소유하고, *PyBuffer_Release()*에 의해 자동으로 해제되고 (즉 참조 횟수가 감소합니다) NULL로 설정됩니다. 이 필드는 표준 C-API 함수의 반환 값과 동등합니다.

특수한 경우로, *PyMemoryView_FromBuffer()* 나 *PyBuffer_FillInfo()*로 감싸진 임시(*temporary*) 버퍼의 경우, 이 필드는 NULL입니다. 일반적으로, 제공하는(exporting) 객체는 이 체계를 사용하지 않아야 합니다.

Py_ssize_t **len**

product(shape) * itemsize. 연속 배열의 경우, 하부 메모리 블록의 길이입니다. 불연속 배열의 경우, 연속 표현으로 복사된다면 논리적 구조체가 갖게 될 길이입니다.

((char *)buf)[0] 에서 ((char *)buf)[len-1] 범위의 액세스는 연속성을 보장하는 요청으로 버퍼가 확보된 경우에만 유효합니다. 대부분 이러한 요청은 *PyBUF_SIMPLE* 또는 *PyBUF_WRITABLE*입니다.

int **readonly**

버퍼가 읽기 전용인지를 나타내는 표시기입니다. 이 필드는 *PyBUF_WRITABLE* 플래그로 제어됩니다.

Py_ssize_t **itemsize**

단일 요소의 항목 크기(바이트)입니다. NULL이 아닌 *format* 값에 호출된 struct.calcsize() 값과 같습니다.

중요한 예외: 소비자가 *PyBUF_FORMAT* 플래그 없이 버퍼를 요청하면, *format*은 NULL로 설정되지만, *itemsize*는 여전히 원래 형식의 값을 갖습니다.

*shape*이 있으면, product(shape) * itemsize == len 동치가 계속 성립하고 소비자는 *itemsize*를 사용하여 버퍼를 탐색할 수 있습니다.

*PyBUF_SIMPLE*이나 *PyBUF_WRITABLE* 요청의 결과로 *shape*이 NULL이면, 소비자는 *itemsize*를 무시하고 itemsize == 1로 가정해야 합니다.

char ***format**

단일 항목의 내용을 설명하는 struct 모듈 스타일 문법의 *NULL* 종료 문자열. 이것이 NULL이면, "B"(부호 없는 바이트)를 가정합니다.

이 필드는 *PyBUF_FORMAT* 플래그로 제어됩니다.

int **ndim**

메모리가 n 차원 배열로 나타내는 차원 수. 0이면, *buf*는 스칼라를 나타내는 단일 항목을 가리킵니다. 이 경우, *shape*, *strides* 및 *suboffsets*는 반드시 NULL 이어야 합니다. 최대 차원 수는 *PyBUF_MAX_NDIM*으로 지정됩니다.

Py_ssize_t ***shape**

n-차원 배열로 메모리의 모양을 나타내는 길이 *ndim*의 *Py_ssize_t* 배열. `shape[0] * ... * shape[ndim-1] * itemsize`는 *len*과 같아야 합니다.

모양 값은 `shape[n] >= 0`로 제한됩니다. `shape[n] == 0`인 경우는 특별한 주의가 필요합니다. 자세한 정보는 복잡한 배열을 참조하십시오.

shape 배열은 소비자에게 읽기 전용입니다.

Py_ssize_t ***strides**

각 차원에서 새 요소를 가져오기 위해 건너뛸 바이트 수를 제공하는 길이 *ndim*의 *Py_ssize_t* 배열.

스트라이드 값은 임의의 정수일 수 있습니다. 일반 배열의 경우, 스트라이드는 보통 양수이지만, 소비자는 `strides[n] <= 0`인 경우를 처리할 수 있어야 합니다. 자세한 내용은 복잡한 배열을 참조하십시오.

strides 배열은 소비자에게 읽기 전용입니다.

Py_ssize_t ***suboffsets**

길이 *ndim*의 *Py_ssize_t* 배열. `suboffsets[n] >= 0` 면, n 번째 차원을 따라 저장된 값은 포인터이고 서브 오프셋 값은 역참조(de-referencing) 후 각 포인터에 더할 바이트 수를 나타냅니다. 음의 서브 오프셋 값은 역참조(de-referencing)가 발생하지 않아야 함을 나타냅니다(연속 메모리 블록에서의 스트라이드).

모든 서브 오프셋이 음수면 (즉, 역참조가 필요하지 않으면), 이 필드는 `NULL`(기본값) 이어야 합니다.

이 유형의 배열 표현은 파이썬 이미징 라이브러리(PIL)에서 사용됩니다. 이러한 배열 요소에 액세스하는 방법에 대한 자세한 내용은 복잡한 배열을 참조하십시오.

suboffsets 배열은 소비자에게 읽기 전용입니다.

void ***internal**

이것은 제공하는(exporting) 객체에 의해 내부적으로 사용됩니다. 예를 들어, 이것은 제공자(exporter)가 정수로 다시 캐스팅할 수 있으며, 버퍼가 해제될 때 shape, strides 및 suboffsets 배열을 해제해야 하는지에 대한 플래그를 저장하는 데 사용됩니다. 소비자가 이 값을 변경해서는 안 됩니다.

상수:

PyBUF_MAX_NDIM

메모리가 나타내는 최대 차원 수 입니다. 제공자는 이 제한을 존중해야 하며, 다차원 버퍼의 소비자는 `PyBUF_MAX_NDIM` 차원까지 처리할 수 있어야 합니다. 현재 64로 설정되었습니다.

8.7.2 버퍼 요청 유형

버퍼는 대개 *PyObject_GetBuffer()*를 통해 제공하는(exporting) 객체로 버퍼 요청을 보내서 얻습니다. 메모리의 논리적 구조의 복잡성이 크게 다를 수 있으므로, 소비자는 처리할 수 있는 정확한 버퍼 유형을 지정하기 위해 *flags* 인자를 사용합니다.

모든 *Py_buffer* 필드는 요청 유형에 의해 모호하지 않게 정의됩니다.

요청 독립적 필드

다음 필드는 *flags*의 영향을 받지 않고 항상 올바른 값으로 채워져야 합니다: *obj*, *buf*, *len*, *itemsize*, *ndim*.

readonly, format

PyBUF_WRITABLE

readonly 필드를 제어합니다. 설정되면, 제공자는 반드시 쓰기 가능한 버퍼를 제공하거나 실패를 보고해야 합니다. 그렇지 않으면, 제공자는 읽기 전용 버퍼나 쓰기 가능 버퍼를

제공할 수 있지만, 모든 소비자에 대해 일관성을 유지해야 합니다. 예를 들어, 간단한 쓰기 가능 버퍼를 요청하려면 *PyBUF_SIMPLE* | *PyBUF_WRITABLE*을 사용할 수 있습니다.

PyBUF_FORMAT

format 필드를 제어합니다. 설정되면, 이 필드를 올바르게 채워야 합니다. 그렇지 않으면, 이 필드는 반드시 NULL 이어야 합니다.

*PyBUF_WRITABLE*은 다음 섹션의 모든 플래그와 | 될 수 있습니다. *PyBUF_SIMPLE*이 0으로 정의되므로, *PyBUF_WRITABLE*은 독립형 플래그로 사용되어 간단한 쓰기 가능한 버퍼를 요청할 수 있습니다.

*PyBUF_FORMAT*은 *PyBUF_SIMPLE*을 제외한 임의의 플래그와 | 되어야 합니다. PyBUF_SIMPLE 은 이미 형식 B(부호 없는 바이트)를 의미하기 때문입니다. PyBUF_FORMAT은 단독으로 사용할 수 없습니다.

shape, strides, suboffsets

메모리의 논리 구조를 제어하는 플래그는 복잡도가 감소하는 순서로 나열됩니다. 각 플래그는 그 아래에 있는 플래그의 모든 비트를 포함합니다.

요청	shape	strides	suboffsets
PyBUF_INDIRECT	yes	yes	필요하면
PyBUF_STRIDES	yes	yes	NULL
PyBUF_ND	yes	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

연속성 요청

C 나 포트란 연속성을 명시적으로 요청할 수 있는데, 스트라이드 정보를 포함하기도 그렇지 않기도 합니다. 스트라이드 정보가 없으면, 버퍼는 C-연속이어야 합니다.

요청	shape	strides	suboffsets	연속성
PyBUF_C_CONTIGUOUS	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS	yes	yes	NULL	C 또는 F
PyBUF_ND	yes	NULL	NULL	C

복합 요청

모든 가능한 요청은 앞 절의 플래그 조합에 의해 완전히 정의됩니다. 편의상, 버퍼 프로토콜은 자주 사용되는 조합을 단일 플래그로 제공합니다.

다음 표에서 *U*는 정의되지 않은 연속성을 나타냅니다. 소비자는 연속성을 판단하기 위해 *PyBuffer_IsContiguous()*를 호출해야 합니다.

요청	shape	strides	suboffsets	연속성	readonly	format
PyBUF_FULL	yes	yes	필요하면	U	0	yes
PyBUF_FULL_RO	yes	yes	필요하면	U	1 또는 0	yes
PyBUF_RECORDS	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO	yes	yes	NULL	U	1 또는 0	yes
PyBUF_STRIDED	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO	yes	yes	NULL	U	1 또는 0	NULL
PyBUF_CONTIG	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	yes	NULL	NULL	C	1 또는 0	NULL

8.7.3 복잡한 배열

NumPy-스타일: shape과 strides

NumPy 스타일 배열의 논리적 구조는 *itemsize*, *ndim*, *shape* 및 *strides*로 정의됩니다.

`ndim == 0`이면, *buf*가 가리키는 메모리 위치가 *itemsize* 크기의 스칼라로 해석됩니다. 이 경우, *shape*과 *strides*는 모두 `NULL`입니다.

*strides*가 `NULL`이면, 배열은 표준 n-차원 C 배열로 해석됩니다. 그렇지 않으면, 소비자는 다음과 같이 n-차원 배열에 액세스해야 합니다:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

위에서 언급했듯이, *buf*는 실제 메모리 블록 내의 모든 위치를 가리킬 수 있습니다. 제공자(exporter)는 이 함수로 버퍼의 유효성을 검사 할 수 있습니다:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """매개 변수가 할당된 메모리 범위 내에서 유효한 배열을 나타내는지 확인합니다:
          char *mem: 물리적 메모리 블록의 시작
          memlen: 물리적 메모리 블록의 길이
          offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

    if ndim <= 0:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        return ndim == 0 and not shape and not strides
    if 0 in shape:
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

PIL-스타일: shape, strides 및 suboffsets

일반 항목 외에도, PIL 스타일 배열에는 차원의 다음 요소를 가져오기 위해 따라야 하는 포인터가 포함될 수 있습니다. 예를 들어, 일반 3-차원 C 배열 `char v[2][2][3]`는 2개의 2-차원 배열을 가리키는 2개의 포인터 배열로 볼 수도 있습니다: `char (*v[2])[2][3]`. suboffsets 표현에서, 이 두 포인터는 *buf*의 시작 부분에 임베드 될 수 있는데, 메모리의 어느 위치 에나 배치될 수 있는 두 개의 `char x[2][3]` 배열을 가리킵니다.

다음은 `NULL`이 아닌 strides와 suboffsets가 있을 때, N-차원 인덱스가 가리키는 N-차원 배열의 요소에 대한 포인터를 반환하는 함수입니다:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

8.7.4 버퍼 관련 함수

int **PyObject_CheckBuffer**(*PyObject* *obj)

Part of the 안정 ABI 버전 *3.11* 이후로. *obj*가 버퍼 인터페이스를 지원하면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. `1`이 반환될 때, *PyObject_GetBuffer()*가 성공할 것이라고 보장하지는 않습니다. 이 함수는 항상 성공합니다.

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int flags)

Part of the 안정 ABI 버전 *3.11* 이후로. *flags*에 지정된 대로 *view*를 채우도록 *exporter*에게 요청을 보냅니다. 제공자(exporter)가 정확한 유형의 버퍼를 제공할 수 없다면, `BufferError`를 일으키고, `view->obj`를 `NULL`로 설정하고, `-1`를 반환해야 합니다.

성공하면, *view*를 채우고, `view->obj`를 *exporter*에 대한 새 참조로 설정하고, 0을 반환합니다. 요청을 단일 객체로 리디렉션하는 연결된(chained) 버퍼 공급자의 경우, `view->obj`는 *exporter* 대신 이 객체를 참조할 수 있습니다 (버퍼 객체 구조체를 보세요).

*PyObject_GetBuffer()*에 대한 성공적인 호출은 *PyBuffer_Release()*에 대한 호출과 쌍을 이루어야 합니다, `malloc()`과 `free()`와 유사합니다. 따라서, 소비자가 버퍼로 작업한 후에는, *PyBuffer_Release()*를 정확히 한 번 호출해야 합니다.

void **PyBuffer_Release**(*Py_buffer* *view)

Part of the 안정 ABI 버전 *3.11* 이후로. 버퍼 *view*를 해제하고 뷰의 지원 객체인 `view->obj`에 대한 강한 참조를 해제합니다 (즉 참조 횟수를 감소시킵니다). 버퍼가 더는 사용되지 않을 때, 이 함수를 반드시 호출해야 합니다. 그렇지 않으면 참조 누수가 발생할 수 있습니다.

*PyObject_GetBuffer()*를 통해 얻지 않은 버퍼에 이 함수를 호출하는 것은 에러입니다.

Py_ssize_t **PyBuffer_SizeFromFormat**(const char *format)

Part of the 안정 ABI 버전 *3.11* 이후로. *format*이 암시하는 *itemsize*를 반환합니다. 에러가 발생하면, 예외를 발생시키고 -1을 반환합니다.

Added in version 3.9.

int **PyBuffer_IsContiguous**(const *Py_buffer* *view, char order)

Part of the 안정 ABI 버전 *3.11* 이후로. *view*로 정의된 메모리가 C 스타일(*order*가 `'C'`)이나 포트란 스타일(*order*가 `'F'`) 연속이거나 둘 중 하나(*order*가 `'A'`)면 `1`을 반환합니다. 그렇지 않으면 `0`을 반환합니다. 이 함수는 항상 성공합니다.

void ***PyBuffer_GetPointer**(const *Py_buffer* *view, const *Py_ssize_t* *indices)

Part of the 안정 ABI 버전 *3.11* 이후로. 주어진 *view* 내부의 *indices*가 가리키는 메모리 영역을 가져옵니다. *indices*는 `view->ndim` 인덱스의 배열을 가리켜야 합니다.

int **PyBuffer_FromContiguous**(const *Py_buffer* *view, const void *buf, *Py_ssize_t* len, char fort)

Part of the 안정 ABI 버전 *3.11* 이후로. *buf*에 있는 연속된 *len* 바이트를 *view*로 복사합니다. *fort*는 `'C'` 또는 `'F'`(C 스타일 또는 포트란 스타일 순서)일 수 있습니다. 성공하면 `0`이 반환되고, 에러가 있으면 `-1`이 반환됩니다.

int **PyBuffer_ToContiguous**(void *buf, const *Py_buffer* *src, *Py_ssize_t* len, char order)

Part of the 안정 ABI 버전 *3.11* 이후로. *src*에 있는 *len* 바이트를 *buf*에 연속 표현으로 복사합니다. *order*는 `'C'` 또는 `'F'` 또는 `'A'`(C 스타일 또는 포트란 스타일 순서 또는 둘 중 하나)일 수 있습니다. 성공하면 `0`이 반환되고, 에러가 있으면 `-1`이 반환됩니다.

이 함수는 *len* != *src->len*이면 실패합니다.

int **PyObject_CopyData**(*PyObject* *dest, *PyObject* *src)

Part of the 안정 ABI 버전 *3.11* 이후로. Copy data from *src* to *dest* buffer. Can convert between C-style and or Fortran-style buffers.

`0` is returned on success, `-1` on error.

void **PyBuffer_FillContiguousStrides**(int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

Part of the 안정 ABI 버전 *3.11* 이후로. *strides* 배열을 주어진 요소당 바이트 수와 주어진 shape 으로 연속 (*order*가 `'C'`면 C 스타일, *order*가 `'F'`면 포트란 스타일) 배열의 바이트 스트라이드로 채웁니다.

int **PyBuffer_FillInfo**(*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

Part of the 안정 ABI 버전 *3.11* 이후로. *readonly*에 따라 쓰기 가능성이 설정된 *len* 크기의 *buf*를 노출하려는 제공자(exporter)에 대한 버퍼 요청을 처리합니다. *buf*는 부호 없는 바이트의 시퀀스로 해석됩니다.

flags 인자는 요청 유형을 나타냅니다. 이 함수는 *buf*가 읽기 전용으로 지정되고 *PyBUF_WRITABLE*이 *flags*에 설정되어 있지 않으면, 항상 플래그가 지정하는 대로 *view*를 채웁니다.

성공하면, `view->obj`를 *exporter*에 대한 새 참조로 설정하고, 0을 반환합니다. 그렇지 않으면, `BufferError`를 일으키고, `view->obj`를 `NULL`로 설정한 다음 `-1`을 반환합니다.

이 함수가 *getbufferproc*의 일부로 사용되면, *exporter*가 제공하는(exporting) 객체로 설정되어야 하고, *flags*는 수정되지 않은 채로 전달되어야 합니다. 그렇지 않으면 *exporter*가 `NULL`이어야 합니다.

CHAPTER 9

구상 객체 계층

이 장의 함수는 특정 파이썬 객체 형에게만 적용됩니다. 그들에게 잘못된 형의 객체를 전달하는 것은 좋은 생각이 아닙니다; 파이썬 프로그램에서 객체를 받았는데 올바른 형을 가졌는지 확실하지 않다면, 먼저 형 검사를 수행해야 합니다; 예를 들어, 객체가 딕셔너리인지 확인하려면, *PyDict_Check()*를 사용하십시오. 이 장은 파이썬 객체 형의 "족보" 처럼 구성되어 있습니다.

경고

이 장에서 설명하는 함수는 전달되는 객체의 형을 주의 깊게 검사하지만, 많은 함수는 유효한 객체 대신 전달되는 NULL을 확인하지 않습니다. NULL을 전달하면 메모리 액세스 위반이 발생하고 인터프리터가 즉시 종료될 수 있습니다.

9.1 기본 객체

이 절에서는 파이썬 형 객체와 싱글톤 객체 None에 대해 설명합니다.

9.1.1 형 객체

type **PyTypeObject**

Part of the 제한된 API (*as an opaque struct*). 내장형을 기술하는 데 사용되는 객체의 C 구조체.

PyTypeObject **PyType_Type**

Part of the 안정 ABI. 이것은 형 객체의 형 객체입니다; 파이썬 계층의 type과 같은 객체입니다.

int **PyType_Check**(*PyObject* *o)

객체 *o*가 표준형 객체에서 파생된 형의 인스턴스를 포함하여 형 객체면 0이 아닌 값을 반환합니다. 다른 모든 경우 0을 반환합니다. 이 함수는 항상 성공합니다.

int **PyType_CheckExact**(*PyObject* *o)

객체 *o*가 형 객체이지만, 표준형 객체의 서브 형이 아니면 0이 아닌 값을 반환합니다. 다른 모든 경우 0을 반환합니다. 이 함수는 항상 성공합니다.

unsigned int **PyType_ClearCache**()

Part of the 안정 ABI. 내부 조회 캐시를 지웁니다. 현재의 버전 태그를 반환합니다.

unsigned long **PyType_GetFlags**(*PyTypeObject* *type)

Part of the 안정 ABI. *type*의 `tp_flags` 멤버를 반환합니다. 이 함수는 주로 `Py_LIMITED_API`와 함께 사용하기 위한 것입니다; 개별 플래그 비트는 파이썬 배포 간에 안정적인 것으로 보장되지만, `tp_flags` 자체에 대한 액세스는 제한된 *API* 일부가 아닙니다.

Added in version 3.2.

버전 3.4에서 변경: 반환형은 이제 `long`이 아니라 `unsigned long`입니다.

PyObject ***PyType_GetDict**(*PyTypeObject* *type)

Return the type object' s internal namespace, which is otherwise only exposed via a read-only proxy (`cls.__dict__`). This is a replacement for accessing `tp_dict` directly. The returned dictionary must be treated as read-only.

This function is meant for specific embedding and language-binding cases, where direct access to the dict is necessary and indirect access (e.g. via the proxy or `PyObject_GetAttr()`) isn' t adequate.

Extension modules should continue to use `tp_dict`, directly or indirectly, when setting up their own types.

Added in version 3.12.

void **PyType_Modified**(*PyTypeObject* *type)

Part of the 안정 ABI. 형과 그것의 모든 서브 형에 대한 내부 검색 캐시를 무효로 합니다. 형의 어트리뷰트나 베이스 클래스를 수동으로 수정한 후에는 이 함수를 호출해야 합니다.

int **PyType_AddWatcher**(*PyType_WatchCallback* callback)

Register *callback* as a type watcher. Return a non-negative integer ID which must be passed to future calls to `PyType_Watch()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

In free-threaded builds, `PyType_AddWatcher()` is not thread-safe, so it must be called at start up (before spawning the first thread).

Added in version 3.12.

int **PyType_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* (previously returned from `PyType_AddWatcher()`). Return `0` on success, `-1` on error (e.g. if *watcher_id* was never registered.)

An extension should never call `PyType_ClearWatcher` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Added in version 3.12.

int **PyType_Watch**(int watcher_id, *PyObject* *type)

Mark *type* as watched. The callback granted *watcher_id* by `PyType_AddWatcher()` will be called whenever `PyType_Modified()` reports a change to *type*. (The callback may be called only once for a series of consecutive modifications to *type*, if `_PyType_Lookup()` is not called on *type* between the modifications; this is an implementation detail and subject to change.)

An extension should never call `PyType_Watch` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Added in version 3.12.

typedef int (***PyType_WatchCallback**)(*PyObject* *type)

Type of a type-watcher callback function.

The callback must not modify *type* or cause `PyType_Modified()` to be called on *type* or any type in its MRO; violating this rule could cause infinite recursion.

Added in version 3.12.

int **PyType_HasFeature**(*PyTypeObject* *o, int feature)

형 객체 *o*가 기능 *feature*를 설정하면 0이 아닌 값을 반환합니다. 형 기능은 단일 비트 플래그로 표시됩니다.

int **PyType_IS_GC**(*PyTypeObject* *o)

형 객체가 순환 검출기에 대한 지원을 포함하고 있으면 참을 반환합니다. 이것은 형 플래그 *Py_TPFLAGS_HAVE_GC*를 검사합니다.

int **PyType_IsSubtype**(*PyTypeObject* *a, *PyTypeObject* *b)

Part of the 안정 ABI. *a*가 *b*의 서브 형이면 참을 반환합니다.

이 함수는 실제 서브 형만 검사합니다. 즉, `__subclasscheck__()`가 *b*에 대해 호출되지 않습니다. `issubclass()`가 수행하는 것과 같은 검사를 하려면 *PyObject_IsSubclass()*를 호출하십시오.

PyObject ***PyType_GenericAlloc**(*PyTypeObject* *type, *Py_ssize_t* nitems)

반환값: 새 참조. *Part of the* 안정 ABI. Generic handler for the *tp_alloc* slot of a type object. Uses Python' s default memory allocation mechanism to allocate memory for a new instance, zeros the memory, then initializes the memory as if by calling *PyObject_Init()* or *PyObject_InitVar()*.

Do not call this directly to allocate memory for an object; call the type' s *tp_alloc* slot instead.

For types that support garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set), this function behaves like *PyObject_GC_New* or *PyObject_GC_NewVar* (except the memory is guaranteed to be zeroed before initialization), and should be paired with *PyObject_GC_Del()* in *tp_free*. Otherwise, it behaves like *PyObject_New* or *PyObject_NewVar* (except the memory is guaranteed to be zeroed before initialization) and should be paired with *PyObject_Free()* in *tp_free*.

PyObject ***PyType_GenericNew**(*PyTypeObject* *type, *PyObject* *args, *PyObject* *kwds)

반환값: 새 참조. *Part of the* 안정 ABI. Generic handler for the *tp_new* slot of a type object. Creates a new instance using the type' s *tp_alloc* slot and returns the resulting object.

int **PyType_Ready**(*PyTypeObject* *type)

Part of the 안정 ABI. 형 개체를 마무리합니다. 초기화를 완료하려면 모든 형 객체에 대해 이 메서드를 호출해야 합니다. 이 함수는 형의 베이스 클래스에서 상속된 슬롯을 추가합니다. 성공 시 `0`을 반환하고, 오류 시 `-1`을 반환하고 예외를 설정합니다.

> **참고**
>
> If some of the base classes implements the GC protocol and the provided type does not include the *Py_TPFLAGS_HAVE_GC* in its flags, then the GC protocol will be automatically implemented from its parents. On the contrary, if the type being created does include *Py_TPFLAGS_HAVE_GC* in its flags then it **must** implement the GC protocol itself by at least implementing the *tp_traverse* handle.

PyObject ***PyType_GetName**(*PyTypeObject* *type)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.11* 이후로. Return the type' s name. Equivalent to getting the type' s `__name__` attribute.

Added in version 3.11.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.11* 이후로. Return the type' s qualified name. Equivalent to getting the type' s `__qualname__` attribute.

Added in version 3.11.

PyObject ***PyType_GetFullyQualifiedName**(*PyTypeObject* *type)

Part of the 안정 ABI 버전 *3.13* 이후로. Return the type' s fully qualified name. Equivalent to `f"{type.__module__}.{type.__qualname__}"`, or `type.__qualname__` if `type.__module__` is not a string or is equal to `"builtins"`.

Added in version 3.13.

PyObject ***PyType_GetModuleName**(*PyTypeObject* *type)

Part of the 안정 ABI 버전 *3.13* 이후로. Return the type' s module name. Equivalent to getting the `type.__module__` attribute.

Added in version 3.13.

void *__PyType_GetSlot__(*PyTypeObject* *type, int slot)

Part of the 안정 ABI 버전 *3.4* 이후로. 지정된 슬롯에 저장된 함수 포인터를 반환합니다. 결과가 NULL 이면, 슬롯이 NULL이거나 함수가 유효하지 않은 매개 변수로 호출되었음을 나타냅니다. 호출자는 일반적으로 결과 포인터를 적절한 함수 형으로 캐스팅합니다.

slot 인자의 가능한 값은 *PyType_Slot.slot*을 참조하십시오.

Added in version 3.4.

버전 3.10에서 변경: *PyType_GetSlot()* can now accept all types. Previously, it was limited to *heap types*.

PyObject *__PyType_GetModule__(*PyTypeObject* *type)

Part of the 안정 ABI 버전 *3.10* 이후로. *PyType_FromModuleAndSpec()*를 사용하여 형을 만들 때 지정된 형과 관련된 모듈 객체를 반환합니다.

주어진 형과 연관된 모듈이 없으면, TypeError를 설정하고 NULL을 반환합니다.

이 함수는 일반적으로 메서드가 정의된 모듈을 가져오는 데 사용됩니다. 이러한 메서드에서, PyType_GetModule(Py_TYPE(self))는 의도한 결과를 반환하지 않을 수 있음에 유의하십시오. Py_TYPE(self)는 의도한 클래스의 서브 클래스일 수 있으며, 서브 클래스가 반드시 슈퍼 클래스와 같은 모듈에 정의된 것은 아닙니다. 메서드를 정의하는 클래스를 얻으려면 *PyCMethod*를 참조하십시오. PyCMethod를 사용할 수 없는 경우는 *PyType_GetModuleByDef()*를 참조하세요.

Added in version 3.9.

void *__PyType_GetModuleState__(*PyTypeObject* *type)

Part of the 안정 ABI 버전 *3.10* 이후로. 주어진 형과 관련된 모듈 객체의 상태를 반환합니다. *PyType_GetModule()* 결과에 *PyModule_GetState()*를 호출하는 바로 가기입니다.

주어진 형과 연관된 모듈이 없으면, TypeError를 설정하고 NULL을 반환합니다.

*type*에 연관된 모듈이 있지만, 상태가 NULL이면, 예외를 설정하지 않고 NULL을 반환합니다.

Added in version 3.9.

PyObject *__PyType_GetModuleByDef__(*PyTypeObject* *type, struct *PyModuleDef* *def)

반환값: 빌린 참조. *Part of the* 안정 ABI 버전 *3.13* 이후로. Find the first superclass whose module was created from the given *PyModuleDef* *def*, and return that module.

모듈을 찾을 수 없으면, TypeError를 발생시키고 NULL을 반환합니다.

This function is intended to be used together with *PyModule_GetState()* to get module state from slot methods (such as *tp_init* or *nb_add*) and other places where a method' s defining class cannot be passed using the *PyCMethod* calling convention.

The returned reference is *borrowed* from *type*, and will be valid as long as you hold a reference to *type*. Do not release it with *Py_DECREF()* or similar.

Added in version 3.11.

int __PyType_GetBaseByToken__(*PyTypeObject* *type, void *token, *PyTypeObject* **result)

Part of the 안정 ABI 버전 *3.14* 이후로. Find the first superclass in *type*' s *method resolution order* whose *Py_tp_token* token is equal to the given one.

- If found, set **result* to a new *strong reference* to it and return 1.
- If not found, set **result* to NULL and return 0.
- On error, set **result* to NULL and return -1 with an exception set.

The *result* argument may be NULL, in which case **result* is not set. Use this if you need only the return value.

The *token* argument may not be NULL.

Added in version 3.14.

int **PyUnstable_Type_AssignVersionTag**(*PyTypeObject* *type)

이것은 불안정 *API*. It may change without warning in minor releases.

Attempt to assign a version tag to the given type.

Returns 1 if the type already had a valid version tag or a new one was assigned, or 0 if a new tag could not be assigned.

Added in version 3.12.

힙에 할당된 형 만들기

다음 함수와 구조체는 힙 형을 만드는 데 사용됩니다.

PyObject ***PyType_FromMetaclass**(*PyTypeObject* *metaclass, *PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Part of the 안정 ABI 버전 *3.12* 이후로. *spec*으로 힙 형을 만들고 반환합니다 (`Py_TPFLAGS_HEAPTYPE`을 참고하세요).

The metaclass *metaclass* is used to construct the resulting type object. When *metaclass* is `NULL`, the metaclass is derived from *bases* (or *Py_tp_base*[*s*] slots if *bases* is `NULL`, see below).

Metaclasses that override `tp_new` are not supported, except if `tp_new` is `NULL`.

베이스 클래스를 지정하는 데 *bases* 인자를 사용할 수 있습니다; 하나의 클래스 또는 클래스의 튜플일 수 있습니다. *bases*가 `NULL`이면, *Py_tp_bases* 슬롯이 대신 사용됩니다. 그 또한 `NULL`이면, *Py_tp_base* 슬롯이 대신 사용됩니다. 그 또한 `NULL`이면, 새 형은 `object`에서 파생됩니다.

module 인자는 새 클래스가 정의된 모듈을 기록하는 데 사용할 수 있습니다. 모듈 객체나 `NULL`이어야 합니다. `NULL`이 아니면, 모듈은 새 형과 연관되며 나중에 `PyType_GetModule()`로 가져올 수 있습니다. 연관된 모듈은 서브 클래스에 상속되지 않습니다; 각 클래스에 대해 개별적으로 지정해야 합니다.

이 함수는 새로운 형에 `PyType_Ready()`를 호출합니다.

Note that this function does *not* fully match the behavior of calling `type()` or using the `class` statement. With user-provided base types or metaclasses, prefer *calling* `type` (or the metaclass) over `PyType_From*` functions. Specifically:

- `__new__()` is not called on the new class (and it must be set to `type.__new__`).
- `__init__()` is not called on the new class.
- `__init_subclass__()` is not called on any bases.
- `__set_name__()` is not called on new descriptors.

Added in version 3.12.

PyObject ***PyType_FromModuleAndSpec**(*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.10* 이후로. `PyType_FromMetaclass(NULL, module, spec, bases)`와 동등합니다.

Added in version 3.9.

버전 3.10에서 변경: The function now accepts a single class as the *bases* argument and `NULL` as the `tp_doc` slot.

버전 3.12에서 변경: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The `tp_new` of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides `tp_new` is deprecated.

버전 3.14에서 변경: Creating classes whose metaclass overrides `tp_new` is no longer allowed.

PyObject ***PyType_FromSpecWithBases**(*PyType_Spec* *spec, *PyObject* *bases)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.3* 이후로. `PyType_FromMetaclass(NULL, NULL, spec, bases)`와 동등합니다.

Added in version 3.3.

버전 3.12에서 변경: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated.

버전 3.14에서 변경: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

PyObject ***PyType_FromSpec**(*PyType_Spec* *spec)

반환값: 새 참조. *Part of the* 안정 ABI. `PyType_FromMetaclass(NULL, NULL, spec, NULL)`와 동등합니다.

버전 3.12에서 변경: The function now finds and uses a metaclass corresponding to the base classes provided in *Py_tp_base*[*s*] slots. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated.

버전 3.14에서 변경: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

int **PyType_Freeze**(*PyTypeObject* *type)

Part of the 안정 ABI 버전 *3.14* 이후로. Make a type immutable: set the *Py_TPFLAGS_IMMUTABLETYPE* flag.

All base classes of *type* must be immutable.

On success, return `0`. On error, set an exception and return `-1`.

The type must not be used before it' s made immutable. For example, type instances must not be created before the type is made immutable.

Added in version 3.14.

type **PyType_Spec**

Part of the 안정 ABI (*including all members*). 형의 행동을 정의하는 구조체.

const char ***name**

형의 이름, *PyTypeObject.tp_name*을 설정하는 데 사용됩니다.

int **basicsize**

양수면, 인스턴스의 바이트 수 크기를 지정합니다. *PyTypeObject.tp_basicsize*를 설정하는 데 사용됩니다.

If zero, specifies that *tp_basicsize* should be inherited.

If negative, the absolute value specifies how much space instances of the class need *in addition* to the superclass. Use *PyObject_GetTypeData()* to get a pointer to subclass-specific memory reserved this way. For negative `basicsize`, Python will insert padding when needed to meet *tp_basicsize*' s alignment requirements.

버전 3.12에서 변경: Previously, this field could not be negative.

int **itemsize**

가변 크기 형의 한 요소의 바이트 수 크기. *PyTypeObject.tp_itemsize*를 설정하는 데 사용됩니다. 주의 사항은 `tp_itemsize` 설명서를 참조하세요.

If zero, *tp_itemsize* is inherited. Extending arbitrary variable-sized classes is dangerous, since some types use a fixed offset for variable-sized memory, which can then overlap fixed-sized memory used by a subclass. To help prevent mistakes, inheriting `itemsize` is only possible in the following situations:

- 베이스는 가변 크기가 아닙니다 (*tp_itemsize* 입니다).

- The requested *PyType_Spec.basicsize* is positive, suggesting that the memory layout of the base class is known.
- The requested *PyType_Spec.basicsize* is zero, suggesting that the subclass does not access the instance' s memory directly.
- With the *Py_TPFLAGS_ITEMS_AT_END* flag.

unsigned int **flags**

형 플래그, *PyTypeObject.tp_flags*를 설정하는 데 사용됩니다.

`Py_TPFLAGS_HEAPTYPE` 플래그가 설정되어 있지 않으면, *PyType_FromSpecWithBases()* 가 자동으로 플래그를 설정합니다.

PyType_Slot ***slots**

PyType_Slot 구조체의 배열. 특수 슬롯값 `{0, NULL}`에 의해 종료됩니다.

Each slot ID should be specified at most once.

type **PyType_Slot**

Part of the 안정 ABI (*including all members*). 형의 선택적 기능을 정의하는 구조체, 슬롯 ID와 값 포인터를 포함합니다.

int **slot**

슬롯 ID.

슬롯 ID는 구조체 *PyTypeObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* 및 *PyAsyncMethods* 의 필드 이름에 `Py_` 접두사를 붙인 이름을 사용합니다. 예를 들어, :

- *PyTypeObject.tp_dealloc*을 설정하는 `Py_tp_dealloc`
- *PyNumberMethods.nb_add*를 설정하는 `Py_nb_add`
- *PySequenceMethods.sq_length*를 설정하는 `Py_sq_length`

An additional slot is supported that does not correspond to a `PyTypeObject` struct field:

- *Py_tp_token*

다음 "offset" 필드는 *PyType_Slot*을 사용하여 설정할 수 없습니다:

- *tp_weaklistoffset* (가능하면 대신 *Py_TPFLAGS_MANAGED_WEAKREF*를 사용하세요)
- *tp_dictoffset* (가능하면 대신 *Py_TPFLAGS_MANAGED_DICT*를 사용하세요)
- *tp_vectorcall_offset* (*PyMemberDef*에서 `"__vectorcalloffset__"`를 사용하세요)

If it is not possible to switch to a `MANAGED` flag (for example, for vectorcall or to support Python older than 3.12), specify the offset in *Py_tp_members*. See *PyMemberDef documentation* for details.

The following internal fields cannot be set at all when creating a heap type:

- *tp_dict*, *tp_mro*, *tp_cache*, *tp_subclasses*, and *tp_weaklist*.

일부 플랫폼에서는 `Py_tp_bases`나 `Py_tp_base`를 설정하는 것이 문제가 될 수 있습니다. 문제를 피하려면, 대신 *PyType_FromSpecWithBases()* 의 *bases* 인자를 사용하십시오.

버전 3.9에서 변경: *PyBufferProcs*의 슬롯은 제한 없는 API에서 설정할 수 있습니다.

버전 3.11에서 변경: *bf_getbuffer* and *bf_releasebuffer* are now available under the *limited API*.

버전 3.14에서 변경: The field *tp_vectorcall* can now set using `Py_tp_vectorcall`. See the field' s documentation for details.

void ***pfunc**

슬롯의 원하는 값입니다. 대부분 이것은 함수에 대한 포인터입니다.

pfunc values may not be `NULL`, except for the following slots:

- Py_tp_doc
- *Py_tp_token* (for clarity, prefer *Py_TP_USE_SPEC* rather than NULL)

Py_tp_token

A *slot* that records a static memory layout ID for a class.

If the *PyType_Spec* of the class is statically allocated, the token can be set to the spec using the special value *Py_TP_USE_SPEC*:

```
static PyType_Slot foo_slots[] = {
   {Py_tp_token, Py_TP_USE_SPEC},
```

It can also be set to an arbitrary pointer, but you must ensure that:

- The pointer outlives the class, so it' s not reused for something else while the class exists.
- It "belongs" to the extension module where the class lives, so it will not clash with other extensions.

Use *PyType_GetBaseByToken()* to check if a class' s superclass has a given token – that is, check whether the memory layout is compatible.

To get the token for a given class (without considering superclasses), use *PyType_GetSlot()* with Py_tp_token.

Added in version 3.14.

Py_TP_USE_SPEC

Used as a value with *Py_tp_token* to set the token to the class' s *PyType_Spec*. Expands to NULL.

Added in version 3.14.

9.1.2 None 객체

None에 대한 *PyTypeObject*는 파이썬/C API에서 직접 노출되지 않습니다. None은 싱글톤이기 때문에 (C에서 ==를 사용해서) 객체 아이덴티티를 검사하는 것으로 충분합니다. 같은 이유로 PyNone_Check() 함수가 없습니다.

PyObject ***Py_None**

값의 부재를 나타내는 파이썬 None 객체입니다. 이 객체는 메서드가 없고 불멸입니다.

버전 3.12에서 변경: *Py_None*은 불멸입니다.

Py_RETURN_NONE

함수에서 *Py_None*을 반환합니다.

9.2 숫자 객체

9.2.1 정수 객체

모든 정수는 임의의 크기의 "long" 정수 객체로 구현됩니다.

에러 시, 대부분의 PyLong_As* API는 숫자와 구별할 수 없는 (return type)-1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

type **PyLongObject**

Part of the 제한된 API (*as an opaque struct*). 이 *PyObject*의 서브 형은 파이썬 정수 객체를 나타냅니다.

PyTypeObject **PyLong_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 정수 형을 나타냅니다. 이것은 파이썬 계층의 int와 같은 객체입니다.

int **PyLong_Check**(*PyObject* *p)

인자가 *PyLongObject*이나 *PyLongObject*의 서브 형이면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyLong_CheckExact**(*PyObject* *p)

인자가 *PyLongObject* 이지만 *PyLongObject*의 서브 형이 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyLong_FromLong**(long v)

반환값: 새 참조. *Part of the* 안정 ABI. *v*로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

CPython 구현 상세: CPython keeps an array of integer objects for all integers between `-5` and `256`. When you create an int in that range you actually just get back a reference to the existing object.

PyObject ***PyLong_FromUnsignedLong**(unsigned long v)

반환값: 새 참조. *Part of the* 안정 ABI. C `unsigned long`으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject ***PyLong_FromSsize_t**(*Py_ssize_t* v)

반환값: 새 참조. *Part of the* 안정 ABI. C *Py_ssize_t*로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject ***PyLong_FromSize_t**(size_t v)

반환값: 새 참조. *Part of the* 안정 ABI. C `size_t`로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject ***PyLong_FromLongLong**(long long v)

반환값: 새 참조. *Part of the* 안정 ABI. C `long long`으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject ***PyLong_FromInt32**(int32_t value)

PyObject ***PyLong_FromInt64**(int64_t value)

Part of the 안정 ABI 버전 *3.14* 이후로. Return a new *PyLongObject* object from a signed C `int32_t` or `int64_t`, or NULL with an exception set on failure.

Added in version 3.14.

PyObject ***PyLong_FromUnsignedLongLong**(unsigned long long v)

반환값: 새 참조. *Part of the* 안정 ABI. C `unsigned long long`으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject ***PyLong_FromUInt32**(uint32_t value)

PyObject ***PyLong_FromUInt64**(uint64_t value)

Part of the 안정 ABI 버전 *3.14* 이후로. Return a new *PyLongObject* object from an unsigned C `uint32_t` or `uint64_t`, or NULL with an exception set on failure.

Added in version 3.14.

PyObject ***PyLong_FromDouble**(double v)

반환값: 새 참조. *Part of the* 안정 ABI. *v*의 정수 부분으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject ***PyLong_FromString**(const char *str, char **pend, int base)

반환값: 새 참조. *Part of the* 안정 ABI. *str*의 문자열 값을 기반으로 한 새 *PyLongObject*를 반환하거나, 실패하면 NULL을 반환합니다. 문자열 값은 *base*의 진수(기수)에 따라 해석됩니다. *pend*가 NULL이 아니면, **pend*는 성공 시에 *str*의 끝을 가리키고, 에러 시에 처리할 수 없었던 첫 번째 문자를 가리킵니다. *base*가 `0`이면, *str*은 integers 정의를 사용해서 해석됩니다; 이때, 0이 아닌 십진수의 선행 0은 `ValueError`를 발생시킵니다. *base*가 `0`이 아니면, `2`와 `36` 사이에 있어야 하며, 경계를 포함합니다. 선행과 후행 공백과 진수 지정자 뒤나 숫자 사이의 단일 밑줄은 무시됩니다. 숫자가 없거나 *str*이 숫자와 후행 공백 뒤에 NULL-종료하지 않으면, `ValueError`가 발생합니다.

> **더 보기**
>
> `PyLong_AsNativeBytes()` and `PyLong_FromNativeBytes()` functions can be used to convert a `PyLongObject` to/from an array of bytes in base `256`.

PyObject ***PyLong_FromUnicodeObject**(*PyObject* *u, int base)

반환값: 새 참조. 문자열 *u*에 있는 유니코드 숫자의 시퀀스를 파이썬 정숫값으로 변환합니다.

Added in version 3.3.

PyObject ***PyLong_FromVoidPtr**(void *p)

반환값: 새 참조. *Part of the* 안정 ABI. 포인터 *p*로부터 파이썬 정수를 만듭니다. 포인터 값은 `PyLong_AsVoidPtr()`를 사용하여 결괏값에서 조회할 수 있습니다.

PyObject ***PyLong_FromNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Part of the 안정 ABI 버전 *3.14* 이후로. Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as a two' s-complement signed number.

flags are as for `PyLong_AsNativeBytes()`. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is a sign bit. Passing `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` will produce the same result as calling `PyLong_FromUnsignedNativeBytes()`. Other flags are ignored.

Added in version 3.13.

PyObject ***PyLong_FromUnsignedNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Part of the 안정 ABI 버전 *3.14* 이후로. Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as an unsigned number.

flags are as for `PyLong_AsNativeBytes()`. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is not a sign bit. Flags other than endian are ignored.

Added in version 3.13.

long **PyLong_AsLong**(*PyObject* *obj)

Part of the 안정 ABI. *obj*의 C `long` 표현을 반환합니다. *obj*가 `PyLongObject`의 인스턴스가 아니면, (있다면) 먼저 `__index__()` 메서드를 호출하여 `PyLongObject`로 변환합니다.

*obj*의 값이 `long`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `-1`을 반환합니다. 모호성을 제거하려면 `PyErr_Occurred()`를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

버전 3.10에서 변경: This function will no longer use `__int__()`.

long **PyLong_AS_LONG**(*PyObject* *obj)

A *soft deprecated* alias. Exactly equivalent to the preferred `PyLong_AsLong`. In particular, it can fail with `OverflowError` or another exception.

버전 3.14부터 폐지됨: The function is soft deprecated.

int **PyLong_AsInt**(*PyObject* *obj)

Part of the 안정 ABI 버전 *3.13* 이후로. Similar to `PyLong_AsLong()`, but store the result in a C `int` instead of a C `long`.

Added in version 3.13.

long **PyLong_AsLongAndOverflow**(*PyObject* *obj, int *overflow)

Part of the 안정 ABI. *obj*의 C `long` 표현을 반환합니다. *obj*가 `PyLongObject`의 인스턴스가 아니면, (있다면) 먼저 `__index__()` 메서드를 호출하여 `PyLongObject`로 변환합니다.

*obj*의 값이 `LONG_MAX`보다 크거나 `LONG_MIN`보다 작으면, **overflow*를 각각 `1`이나 `-1`로 설정하고 `-1`을 반환합니다; 그렇지 않으면, **overflow*를 `0`으로 설정합니다. 다른 예외가 발생하면 **overflow*를 `0`으로 설정하고 `-1`을 평소와 같이 반환합니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 __index__()를 사용합니다.

버전 3.10에서 변경: This function will no longer use __int__().

long long **PyLong_AsLongLong**(*PyObject* *obj)

Part of the 안정 ABI. *obj*의 C long long 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 __index__() 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 long long의 범위를 벗어나면 OverflowError를 발생시킵니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 __index__()를 사용합니다.

버전 3.10에서 변경: This function will no longer use __int__().

long long **PyLong_AsLongLongAndOverflow**(*PyObject* *obj, int *overflow)

Part of the 안정 ABI. *obj*의 C long long 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 __index__() 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 LLONG_MAX보다 크거나 LLONG_MIN보다 작으면, **overflow*를 각각 1이나 -1로 설정하고 -1을 반환합니다; 그렇지 않으면, **overflow*를 0으로 설정합니다. 다른 예외가 발생하면 **overflow*를 0으로 설정하고 -1을 평소와 같이 반환합니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

Added in version 3.2.

버전 3.8에서 변경: 사용할 수 있으면 __index__()를 사용합니다.

버전 3.10에서 변경: This function will no longer use __int__().

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* *pylong)

Part of the 안정 ABI. *pylong*의 C *Py_ssize_t* 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 *Py_ssize_t*의 범위를 벗어나면 OverflowError를 발생시킵니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

unsigned long **PyLong_AsUnsignedLong**(*PyObject* *pylong)

Part of the 안정 ABI. *pylong*의 C unsigned long 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 unsigned long의 범위를 벗어나면 OverflowError를 발생시킵니다.

에러 시 (unsigned long)-1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

size_t **PyLong_AsSize_t**(*PyObject* *pylong)

Part of the 안정 ABI. *pylong*의 C size_t 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 size_t의 범위를 벗어나면 OverflowError를 발생시킵니다.

에러 시 (size_t)-1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* *pylong)

Part of the 안정 ABI. *pylong*의 C unsigned long long 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 unsigned long long의 범위를 벗어나면 OverflowError를 발생시킵니다.

에러 시 (unsigned long long)-1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.1에서 변경: 음의 *pylong*는 이제 TypeError가 아니라 OverflowError를 발생시킵니다.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

Part of the 안정 ABI. *obj*의 C `unsigned long` 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 `__index__()` 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 `unsigned long`의 범위를 벗어나면, 그 값의 모듈로 `ULONG_MAX + 1` 환원을 반환합니다.

에러 시 `(unsigned long)-1`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

버전 3.10에서 변경: This function will no longer use `__int__()`.

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* *obj)

Part of the 안정 ABI. *obj*의 C `unsigned long long` 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 `__index__()` 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 `unsigned long long`의 범위를 벗어나면, 그 값의 모듈로 `ULLONG_MAX + 1` 환원을 반환합니다.

에러 시 `(unsigned long long)-1`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

버전 3.10에서 변경: This function will no longer use `__int__()`.

int **PyLong_AsInt32**(*PyObject* *obj, int32_t *value)

int **PyLong_AsInt64**(*PyObject* *obj, int64_t *value)

Part of the 안정 ABI 버전 *3.14* 이후로. Set **value* to a signed C `int32_t` or `int64_t` representation of *obj*.

If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the *obj* value is out of range, raise an `OverflowError`.

Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Added in version 3.14.

int **PyLong_AsUInt32**(*PyObject* *obj, uint32_t *value)

int **PyLong_AsUInt64**(*PyObject* *obj, uint64_t *value)

Part of the 안정 ABI 버전 *3.14* 이후로. Set **value* to an unsigned C `uint32_t` or `uint64_t` representation of *obj*.

If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

- If *obj* is negative, raise a `ValueError`.
- If the *obj* value is out of range, raise an `OverflowError`.

Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Added in version 3.14.

double **PyLong_AsDouble**(*PyObject* *pylong)

Part of the 안정 ABI. *pylong*의 C `double` 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 `double`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `-1.0`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

void *__PyLong_AsVoidPtr__(*PyObject* *pylong)

Part of the 안정 ABI. 파이썬 정수 *pylong*을 C `void` 포인터로 변환합니다. *pylong*을 변환할 수 없으면, `OverflowError`가 발생합니다. 이것은 *PyLong_FromVoidPtr()*로 만들어진 값에 대해서만 사용할 수 있는 `void` 포인터를 생성하는 것이 보장됩니다.

에러 시 `NULL`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

Py_ssize_t __PyLong_AsNativeBytes__(*PyObject* *pylong, void *buffer, *Py_ssize_t* n_bytes, int flags)

Part of the 안정 ABI 버전 *3.14* 이후로. Copy the Python integer value *pylong* to a native *buffer* of size *n_bytes*. The *flags* can be set to `-1` to behave similarly to a C cast, or to values documented below to control the behavior.

Returns `-1` with an exception raised on error. This may happen if *pylong* cannot be interpreted as an integer, or if *pylong* was negative and the `Py_ASNATIVEBYTES_REJECT_NEGATIVE` flag was set.

Otherwise, returns the number of bytes required to store the value. If this is equal to or less than *n_bytes*, the entire value was copied. All *n_bytes* of the buffer are written: large buffers are padded with zeroes.

If the returned value is greater than *n_bytes*, the value was truncated: as many of the lowest bits of the value as could fit are written, and the higher bits are ignored. This matches the typical behavior of a C-style downcast.

> **참고**
>
> Overflow is not considered an error. If the returned value is larger than *n_bytes*, most significant bits were discarded.

`0` will never be returned.

Values are always copied as two' s-complement.

Usage example:

```
int32_t value;
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, &value, sizeof(value), -1);
if (bytes < 0) {
    // 실패. 이유와 함께 파이썬 예외가 설정됩니다.
    return NULL;
}
else if (bytes <= (Py_ssize_t)sizeof(value)) {
    // 성공!
}
else {
    // 오버플로우가 발생했지만, 'value' 에 pylong 의 잘린
    // 하위 비트들이 담깁니다.
}
```

Passing zero to *n_bytes* will return the size of a buffer that would be large enough to hold the value. This may be larger than technically necessary, but not unreasonably so. If *n_bytes=0*, *buffer* may be `NULL`.

> **참고**
>
> Passing *n_bytes=0* to this function is not an accurate way to determine the bit length of the value.

To get at the entire Python value of an unknown size, the function can be called twice: first to determine the buffer size, then to fill it:

```
// 필요한 공간의 크기를 물어봅니다.
Py_ssize_t expected = PyLong_AsNativeBytes(pylong, NULL, 0, -1);
if (expected < 0) {
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    // 실패. 이유와 함께 파이썬 예외가 설정됩니다.
    return NULL;
}
assert(expected != 0);  // API 정의에 따라 불가능합니다.
uint8_t *bignum = malloc(expected);
if (!bignum) {
    PyErr_SetString(PyExc_MemoryError, "bignum malloc failed.");
    return NULL;
}
// 안전하게 전체 값을 얻습니다.
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, bignum, expected, -1);
if (bytes < 0) {  // 예외가 설정되었습니다.
    free(bignum);
    return NULL;
}
else if (bytes > expected) {  // 이는 불가능해야 합니다.
    PyErr_SetString(PyExc_RuntimeError,
        "Unexpected bignum truncation after a size check.");
    free(bignum);
    return NULL;
}
// 위의 사전 점검을 고려할 때 예상되는 성공.
// ... bignum 을 사용합니다 ...
free(bignum);
```

flags is either `-1` (`Py_ASNATIVEBYTES_DEFAULTS`) to select defaults that behave most like a C cast, or a combination of the other flags in the table below. Note that `-1` cannot be combined with other flags.

Currently, `-1` corresponds to `Py_ASNATIVEBYTES_NATIVE_ENDIAN | Py_ASNATIVEBYTES_UNSIGNED_BUFFER`.

플래그	값
Py_ASNATIVEBYTES_DEFAULTS	-1
Py_ASNATIVEBYTES_BIG_ENDIAN	0
Py_ASNATIVEBYTES_LITTLE_ENDIAN	1
Py_ASNATIVEBYTES_NATIVE_ENDIAN	3
Py_ASNATIVEBYTES_UNSIGNED_BUFFER	4
Py_ASNATIVEBYTES_REJECT_NEGATIVE	8
Py_ASNATIVEBYTES_ALLOW_INDEX	16

Specifying `Py_ASNATIVEBYTES_NATIVE_ENDIAN` will override any other endian flags. Passing `2` is reserved.

By default, sufficient buffer will be requested to include a sign bit. For example, when converting 128 with

n_bytes=1, the function will return 2 (or more) in order to store a zero sign bit.

If `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` is specified, a zero sign bit will be omitted from size calculations. This allows, for example, 128 to fit in a single-byte buffer. If the destination buffer is later treated as signed, a positive input value may become negative. Note that the flag does not affect handling of negative values: for those, space for a sign bit is always requested.

Specifying `Py_ASNATIVEBYTES_REJECT_NEGATIVE` causes an exception to be set if *pylong* is negative. Without this flag, negative values will be copied provided there is enough space for at least one sign bit, regardless of whether `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` was specified.

If `Py_ASNATIVEBYTES_ALLOW_INDEX` is specified and a non-integer value is passed, its `__index__()` method will be called first. This may result in Python code executing and other threads being allowed to run, which could cause changes to other objects or values in use. When *flags* is `-1`, this option is not set, and non-integer values will raise `TypeError`.

> **참고**
>
> With the default *flags* (`-1`, or *UNSIGNED_BUFFER* without *REJECT_NEGATIVE*), multiple Python integers can map to a single value without overflow. For example, both `255` and `-1` fit a single-byte buffer and set all its bits. This matches typical C cast behavior.

Added in version 3.13.

int **PyLong_GetSign**(*PyObject* *obj, int *sign)

Get the sign of the integer object *obj*.

On success, set **sign* to the integer sign (0, -1 or +1 for zero, negative or positive integer, respectively) and return 0.

On failure, return -1 with an exception set. This function always succeeds if *obj* is a `PyLongObject` or its subtype.

Added in version 3.14.

int **PyLong_IsPositive**(*PyObject* *obj)

Check if the integer object *obj* is positive (`obj > 0`).

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it' s positive and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

int **PyLong_IsNegative**(*PyObject* *obj)

Check if the integer object *obj* is negative (`obj < 0`).

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it' s negative and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

int **PyLong_IsZero**(*PyObject* *obj)

Check if the integer object *obj* is zero.

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it' s zero and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

PyObject ***PyLong_GetInfo**(void)

Part of the 안정 ABI. On success, return a read only *named tuple*, that holds information about Python' s internal representation of integers. See `sys.int_info` for description of individual fields.

On failure, return `NULL` with an exception set.

Added in version 3.1.

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)

이것은 불안정 *API*. It may change without warning in minor releases.

Return 1 if *op* is compact, 0 otherwise.

This function makes it possible for performance-critical code to implement a "fast path" for small integers. For compact values use `PyUnstable_Long_CompactValue()`; for others fall back to a `PyLong_As*` function or `PyLong_AsNativeBytes()`.

The speedup is expected to be negligible for most users.

Exactly what values are considered compact is an implementation detail and is subject to change.

Added in version 3.12.

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)

이것은 불안정 *API*. It may change without warning in minor releases.

If *op* is compact, as determined by `PyUnstable_Long_IsCompact()`, return its value.

Otherwise, the return value is undefined.

Added in version 3.12.

Export API

Added in version 3.14.

struct **PyLongLayout**

Layout of an array of "digits" ("limbs" in the GMP terminology), used to represent absolute value for arbitrary precision integers.

Use `PyLong_GetNativeLayout()` to get the native layout of Python `int` objects, used internally for integers with "big enough" absolute value.

See also `sys.int_info` which exposes similar information in Python.

uint8_t **bits_per_digit**

Bits per digit. For example, a 15 bit digit means that bits 0-14 contain meaningful information.

uint8_t **digit_size**

Digit size in bytes. For example, a 15 bit digit will require at least 2 bytes.

int8_t **digits_order**

Digits order:

- `1` for most significant digit first
- `-1` for least significant digit first

int8_t **digit_endianness**

Digit endianness:

- `1` for most significant byte first (big endian)
- `-1` for least significant byte first (little endian)

const *PyLongLayout* ***PyLong_GetNativeLayout**(void)

Get the native layout of Python `int` objects.

See the *PyLongLayout* structure.

The function must not be called before Python initialization nor after Python finalization. The returned layout is valid until Python is finalized. The layout is the same for all Python sub-interpreters in a process, and so it can be cached.

struct **PyLongExport**

Export of a Python `int` object.

There are two cases:

- If *digits* is `NULL`, only use the *value* member.
- If *digits* is not `NULL`, use *negative*, *ndigits* and *digits* members.

int64_t **value**

The native integer value of the exported `int` object. Only valid if *digits* is `NULL`.

uint8_t **negative**

`1` if the number is negative, `0` otherwise. Only valid if *digits* is not `NULL`.

Py_ssize_t **ndigits**

Number of digits in *digits* array. Only valid if *digits* is not `NULL`.

const void ***digits**

Read-only array of unsigned digits. Can be `NULL`.

int **PyLong_Export**(*PyObject* *obj, *PyLongExport* *export_long)

Export a Python `int` object.

export_long must point to a *PyLongExport* structure allocated by the caller. It must not be `NULL`.

On success, fill in **export_long* and return `0`. On error, set an exception and return `-1`.

PyLong_FreeExport() must be called when the export is no longer needed.

CPython 구현 상세: This function always succeeds if *obj* is a Python `int` object or a subclass.

void **PyLong_FreeExport**(*PyLongExport* *export_long)

Release the export *export_long* created by *PyLong_Export()*.

CPython 구현 상세: Calling *PyLong_FreeExport()* is optional if *export_long->digits* is `NULL`.

PyLongWriter API

The *PyLongWriter* API can be used to import an integer.

Added in version 3.14.

struct **PyLongWriter**

A Python `int` writer instance.

The instance must be destroyed by *PyLongWriter_Finish()* or *PyLongWriter_Discard()*.

PyLongWriter ***PyLongWriter_Create**(int negative, *Py_ssize_t* ndigits, void **digits)

Create a *PyLongWriter*.

On success, allocate **digits* and return a writer. On error, set an exception and return `NULL`.

negative is `1` if the number is negative, or `0` otherwise.

ndigits is the number of digits in the *digits* array. It must be greater than 0.

digits must not be NULL.

After a successful call to this function, the caller should fill in the array of digits *digits* and then call *PyLongWriter_Finish()* to get a Python `int`. The layout of *digits* is described by *PyLong_GetNativeLayout()*.

Digits must be in the range `[0; (1 << bits_per_digit) - 1]` (where the *bits_per_digit* is the number of bits per digit). Any unused most significant digits must be set to `0`.

Alternately, call *PyLongWriter_Discard()* to destroy the writer instance without creating an `int` object.

PyObject ***PyLongWriter_Finish**(*PyLongWriter* *writer)

반환값: 새 참조. Finish a *PyLongWriter* created by *PyLongWriter_Create()*.

On success, return a Python `int` object. On error, set an exception and return `NULL`.

The function takes care of normalizing the digits and converts the object to a compact integer if needed.

The writer instance and the *digits* array are invalid after the call.

void **PyLongWriter_Discard**(*PyLongWriter* *writer)

Discard a *PyLongWriter* created by *PyLongWriter_Create()*.

If *writer* is `NULL`, no operation is performed.

The writer instance and the *digits* array are invalid after the call.

9.2.2 불리언 객체

파이썬에서 불리언은 정수의 서브 클래스로 구현됩니다. *Py_False*와 *Py_True*라는 두 개의 부울만 있습니다. 따라서 일반적인 생성 및 삭제 함수는 부울에 적용되지 않습니다. 그러나 다음 매크로를 사용할 수 있습니다.

PyTypeObject **PyBool_Type**

Part of the 안정 ABI. This instance of *PyTypeObject* represents the Python boolean type; it is the same object as `bool` in the Python layer.

int **PyBool_Check**(*PyObject* *o)

*o*가 *PyBool_Type* 형이면 참을 돌려줍니다. 이 함수는 항상 성공합니다.

PyObject ***Py_False**

파이썬 `False` 객체. 이 객체는 메서드가 없고 불멸입니다.

버전 3.12에서 변경: *Py_False*는 불멸입니다.

PyObject ***Py_True**

파이썬 `True` 객체. 이 객체는 메서드가 없고 불멸입니다.

버전 3.12에서 변경: *Py_True*는 불멸입니다.

Py_RETURN_FALSE

함수에서 *Py_False*를 반환합니다.

Py_RETURN_TRUE

함수에서 *Py_True*를 반환합니다.

PyObject ***PyBool_FromLong**(long v)

반환값: 새 참조. *Part of the* 안정 ABI. *v*의 논리값에 따라 *Py_True* 나 *Py_False*를 반환합니다.

9.2.3 부동 소수점 객체

type **PyFloatObject**

이 *PyObject*의 서브 형은 파이썬 부동 소수점 객체를 나타냅니다.

PyTypeObject **PyFloat_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 부동 소수점 형을 나타냅니다. 이것은 파이썬 계층에서 `float`와 같은 객체입니다.

int **PyFloat_Check**(*PyObject* *p)

인자가 `PyFloatObject` 나 `PyFloatObject`의 서브 형이면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyFloat_CheckExact**(*PyObject* *p)

인자가 `PyFloatObject`이지만 `PyFloatObject`의 서브 형은 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyFloat_FromString**(*PyObject* *str)

반환값: 새 참조. *Part of the* 안정 ABI. *str*의 문자열 값을 기반으로 `PyFloatObject` 객체를 만들거나, 실패하면 `NULL`.

PyObject ***PyFloat_FromDouble**(double v)

반환값: 새 참조. *Part of the* 안정 ABI. *v*로부터 `PyFloatObject` 객체를 만들거나, 실패하면 `NULL`.

double **PyFloat_AsDouble**(*PyObject* *pyfloat)

Part of the 안정 ABI. *pyfloat* 의 내용의 C `double` 표현을 반환합니다. *pyfloat*가 파이썬 부동 소수점 객체가 아니지만 `__float__()` 메서드가 있으면, *pyfloat*를 float로 변환하기 위해 이 메서드가 먼저 호출됩니다. `__float__()`가 정의되지 않았으면 `__index__()`로 대체합니다. 이 메서드는 실패하면 `-1.0`을 반환하므로, `PyErr_Occurred()`를 호출하여 에러를 확인해야 합니다.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)

에러 검사 없이 *pyfloat* 의 내용의 C `double` 표현을 반환합니다.

PyObject ***PyFloat_GetInfo**(void)

반환값: 새 참조. *Part of the* 안정 ABI. float의 정밀도, 최솟값, 최댓값에 관한 정보를 포함한 structseq 인스턴스를 돌려줍니다. 헤더 파일 `float.h`를 감싸는 얇은 래퍼입니다.

double **PyFloat_GetMax**()

Part of the 안정 ABI. 최대 표현 가능한 유한 float *DBL_MAX*를 C `double`로 반환합니다.

double **PyFloat_GetMin**()

Part of the 안정 ABI. 최소 정규화된(normalized) 양의 float *DBL_MIN*를 C `double`로 반환합니다.

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C `double`, and the Unpack routines produce a C `double` from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn' t handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

Note that NaNs type may not be preserved on IEEE platforms (silent NaN become quiet), for example on x86 systems in 32-bit mode.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

Added in version 3.11.

Pack functions

The pack routines write 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if you want the bytes string in little-endian format (exponent last, at `p+1`, `p+3`, or `p+6` `p+7`), zero if you want big-endian format (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: `0` if all is OK, `-1` if error (and an exception is set, most likely `OverflowError`).

There are two problems on non-IEEE platforms:

- What this does is undefined if *x* is a NaN or infinity.
- `-0.0` and `+0.0` produce the same bytes string.

int **PyFloat_Pack2**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary16 half-precision format.

int **PyFloat_Pack4**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary32 single precision format.

int **PyFloat_Pack8**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: The unpacked double. On error, this is `-1.0` and *`PyErr_Occurred()`* is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

double **PyFloat_Unpack2**(const char *p, int le)

Unpack the IEEE 754 binary16 half-precision format as a C double.

double **PyFloat_Unpack4**(const char *p, int le)

Unpack the IEEE 754 binary32 single precision format as a C double.

double **PyFloat_Unpack8**(const char *p, int le)

Unpack the IEEE 754 binary64 double precision format as a C double.

9.2.4 복소수 객체

파이썬의 복소수 객체는 C API에서 볼 때 두 개의 다른 형으로 구현됩니다: 하나는 파이썬 프로그램에 노출된 파이썬 객체이고, 다른 하나는 실제 복소수 값을 나타내는 C 구조체입니다. API는 두 가지 모두도 작업할 수 있는 함수를 제공합니다.

C 구조체로서의 복소수

매개 변수로 이러한 구조체를 받아들이고 결과로 반환하는 함수는 포인터를 통해 역참조하기보다는 값으로 다룹니다. 이는 API 전체에서 일관됩니다.

type **Py_complex**

파이썬 복소수 객체의 값 부분에 해당하는 C 구조체. 복소수 객체를 다루는 대부분 함수는 이 형의 구조체를 입력 또는 출력값으로 적절하게 사용합니다.

double **real**

double **imag**

구조체는 다음과 같이 정의됩니다:

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

Py_complex **_Py_c_sum**(*Py_complex* left, *Py_complex* right)

C *Py_complex* 표현을 사용하여 두 복소수의 합을 반환합니다.

Py_complex **_Py_c_diff**(*Py_complex* left, *Py_complex* right)

C *Py_complex* 표현을 사용하여 두 복소수의 차이를 반환합니다.

Py_complex **_Py_c_neg**(*Py_complex* num)

C *Py_complex* 표현을 사용하여 복소수 *num*의 음의 값을 반환합니다.

Py_complex **_Py_c_prod**(*Py_complex* left, *Py_complex* right)

C *Py_complex* 표현을 사용하여 두 복소수의 곱을 반환합니다.

Py_complex **_Py_c_quot**(*Py_complex* dividend, *Py_complex* divisor)

C *Py_complex* 표현을 사용하여 두 복소수의 몫을 반환합니다.

*divisor*가 null이면, 이 메서드는 0을 반환하고, `errno`를 `EDOM`으로 설정합니다.

Py_complex **_Py_c_pow**(*Py_complex* num, *Py_complex* exp)

C *Py_complex* 표현을 사용하여 *num*의 *exp* 거듭제곱을 반환합니다.

*num*이 null이고 *exp*가 양의 실수가 아니면, 이 메서드는 0을 반환하고 `errno`를 `EDOM`으로 설정합니다.

Set `errno` to `ERANGE` on overflows.

파이썬 객체로서의 복소수

type **PyComplexObject**

파이썬 복소수 객체를 나타내는 *PyObject*의 서브 형.

PyTypeObject **PyComplex_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 복소수 형을 나타냅니다. 파이썬 계층의 `complex`와 같은 객체입니다.

int **PyComplex_Check**(*PyObject* *p)

인자가 *PyComplexObject* 나 *PyComplexObject*의 서브 형이면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyComplex_CheckExact**(*PyObject* *p)

인자가 *PyComplexObject*이지만, *PyComplexObject*의 서브 유형이 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyComplex_FromCComplex**(*Py_complex* v)

반환값: 새 참조. C *Py_complex* 값으로 새로운 파이썬 복소수 객체를 만듭니다. 에러면 예외를 설정하고 `NULL`을 반환합니다.

PyObject ***PyComplex_FromDoubles**(double real, double imag)

반환값: 새 참조. *Part of the* 안정 ABI. *real* 및 *imag*로 새로운 *PyComplexObject* 객체를 반환합니다. 에러면 예외를 설정하고 `NULL`을 반환합니다.

double **PyComplex_RealAsDouble**(*PyObject* *op)

Part of the 안정 ABI. *op*의 실수부를 C `double`로 반환합니다.

*op*가 파이썬 복소수 객체가 아니지만 `__complex__()` 메서드가 있으면, 이 메서드는 먼저 *op*를 파이썬 복소수 객체로 변환하도록 그 메서드를 호출합니다. `__complex__()`가 정의되지 않았으면 *PyFloat_AsDouble()*을 호출하는 것으로 대체하고 그 결과를 반환합니다.

실패하면, 이 메서드는 예외를 설정하고 `-1.0`을 반환합니다. *PyErr_Occurred()*를 호출하여 에러를 확인해야 합니다.

버전 3.13에서 변경: 사용할 수 있다면 `__complex__()`를 사용합니다.

double **PyComplex_ImagAsDouble**(*PyObject* *op)

Part of the 안정 ABI. *op*의 허수부를 C `double`로 반환합니다.

*op*가 파이썬 복소수 객체가 아니지만 `__complex__()` 메서드가 있으면, 이 메서드는 먼저 *op*를 파이썬 복소수 객체로 변환하도록 그 메서드를 호출합니다. `__complex__()`가 정의되지 않았으면 *PyFloat_AsDouble()*을 호출하는 것으로 대체하고 성공할 때 `0.0`을 반환합니다.

실패하면, 이 메서드는 예외를 설정하고 `-1.0`을 반환합니다. *PyErr_Occurred()*를 호출하여 에러를 확인해야 합니다.

버전 3.13에서 변경: 사용할 수 있다면 `__complex__()`를 사용합니다.

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

복소수 *op*의 *Py_complex* 값을 반환합니다.

*op*가 파이썬 복소수 객체가 아니지만 `__complex__()` 메서드가 있으면, 이 메서드는 먼저 *op*를 파이썬 복소수 객체로 변환하도록 그 메서드를 호출합니다. `__complex__()`가 정의되지 않았으면 `__float__()`로 대체합니다. `__float__()`가 정의되지 않았으면 `__index__()`로 대체합니다.

실패하면, 이 메서드는 예외를 설정하고 *real*을 `-1.0`으로 설정한 *Py_complex*를 반환합니다. *PyErr_Occurred()*를 호출하여 에러를 확인해야 합니다.

버전 3.8에서 변경: 사용할 수 있다면 `__index__()`를 사용합니다.

9.3 시퀀스 객체

시퀀스 객체에 대한 일반적인 연산은 이전 장에서 논의했습니다; 이 절에서는 파이썬 언어에 고유한 특정 종류의 시퀀스 객체를 다룹니다.

9.3.1 바이트열 객체

이 함수들은 바이트열 매개 변수가 필요할 때 바이트열이 아닌 매개 변수로 호출하면 `TypeError`를 발생시킵니다.

type **PyBytesObject**

이 *PyObject*의 서브 형은 파이썬 바이트열 객체를 나타냅니다.

PyTypeObject **PyBytes_Type**

Part of the 안정 ABI. 이 *PyTypeObject*의 인스턴스는 파이썬 바이트열 형을 나타냅니다; 파이썬 계층의 `bytes`와 같은 객체입니다.

int **PyBytes_Check**(*PyObject* *o)

객체 *o*가 바이트열 객체이거나 바이트열 형의 서브 형의 인스턴스면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyBytes_CheckExact**(*PyObject* *o)

객체 *o*가 바이트열 객체이지만, 바이트열 형의 서브 형의 인스턴스는 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyBytes_FromString**(const char *v)

반환값: 새 참조. *Part of the* 안정 ABI. 성공하면 값으로 *v* 문자열의 복사본을 갖는 새 바이트열 객체를 반환하고, 실패하면 `NULL`을 반환합니다. 매개 변수 *v*는 `NULL`이 아니어야 합니다; 검사하지 않습니다.

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

반환값: 새 참조. *Part of the* 안정 ABI. 성공하면 값이 *v* 문자열의 복사본이고 길이가 *len*인 새 바이트열 객체를 반환하고, 실패하면 `NULL`을 반환합니다. *v*가 `NULL`이면, 바이트열 객체의 내용은 초기화되지 않습니다.

PyObject ***PyBytes_FromFormat**(const char *format, ...)

반환값: 새 참조. *Part of the* 안정 ABI. C `printf()`-스타일 *format* 문자열과 가변 개수의 인자를 받아서, 결과 파이썬 바이트열 객체의 크기를 계산하고 그 안에 값이 포맷된 바이트열 객체를 반환합니다.

가변 인자는 C 형이어야 하며 *format* 문자열에 있는 포맷 문자들과 정확히 대응해야 합니다. 허용되는 포맷 문자는 다음과 같습니다:

포맷 문자	형	주석
`%%`	*n/a*	리터럴 % 문자.
`%c`	int	단일 바이트, C int로 표현됩니다.
`%d`	int	`printf("%d")`와 동등합니다.[1]
`%u`	unsigned int	`printf("%u")`와 동등합니다.[1]
`%ld`	long	`printf("%ld")`와 동등합니다.[1]
`%lu`	unsigned long	`printf("%lu")`와 동등합니다.[1]
`%zd`	*Py_ssize_t*	`printf("%zd")`와 동등합니다.[1]
`%zu`	size_t	`printf("%zu")`와 동등합니다.[1]
`%i`	int	`printf("%i")`와 동등합니다.[1]
`%x`	int	`printf("%x")`와 동등합니다.[1]
`%s`	const char*	널-종료 C 문자 배열.
`%p`	const void*	C 포인터의 16진수 표현. 플랫폼의 `printf`가 어떤 결과를 내는지에 상관없이 리터럴 `0x`로 시작함이 보장된다는 점을 제외하고는 거의 `printf("%p")`와 동등합니다.

인식할 수 없는 포맷 문자는 포맷 문자열의 나머지 부분이 모두 결과 객체에 그대로 복사되게 만들고, 추가 인자는 무시됩니다.

PyObject ***PyBytes_FromFormatV**(const char *format, va_list vargs)

반환값: 새 참조. *Part of the* 안정 ABI. 정확히 두 개의 인자를 취한다는 것을 제외하고는 *PyBytes_FromFormat()*과 같습니다.

PyObject ***PyBytes_FromObject**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 버퍼 프로토콜을 구현하는 객체 *o*의 바이트열 표현을 반환합니다.

Py_ssize_t **PyBytes_Size**(*PyObject* *o)

Part of the 안정 ABI. 바이트열 객체 *o*의 길이를 반환합니다.

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)

*PyBytes_Size()*와 유사하지만, 에러 검사가 없습니다.

char ***PyBytes_AsString**(*PyObject* *o)

Part of the 안정 ABI. *o*의 내용에 대한 포인터를 반환합니다. 포인터는 `len(o) + 1` 바이트로 구성된 *o*의 내부 버퍼를 가리킵니다. 버퍼의 마지막 바이트는 다른 널(null) 바이트가 있는지에 관계없이 항상 널입니다. 객체가 `PyBytes_FromStringAndSize(NULL, size)`를 사용하여 방금 만들어진 경우가 아니면 데이터를 수정해서는 안 됩니다. 할당을 해제해서는 안 됩니다. *o*가 바이트열 객체가 아니면, *PyBytes_AsString()*은 `NULL`을 반환하고 `TypeError`를 발생시킵니다.

char ***PyBytes_AS_STRING**(*PyObject* *string)

*PyBytes_AsString()*와 유사하지만, 에러 검사가 없습니다.

int **PyBytes_AsStringAndSize**(*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

Part of the 안정 ABI. 출력 변수 *buffer*와 *length*로 객체 *obj*의 널-종료 내용을 반환합니다. 성공하면 `0`을 반환합니다.

*length*가 `NULL`이면, 바이트열 객체는 내장된 널 바이트를 포함할 수 없습니다; 만약 그렇다면 함수는 `-1`을 반환하고 `ValueError`를 발생시킵니다.

buffer는 *obj*의 내부 버퍼를 가리키게 되는데, 끝에 추가 널 바이트가 포함됩니다 (*length*에는 포함되지 않습니다). 객체가 `PyBytes_FromStringAndSize(NULL, size)`를 사용하여 방금 만들어진 경우가 아니면 데이터를 수정해서는 안 됩니다. 할당을 해제해서는 안 됩니다. *obj*가 바이트열 객체가 아니면 *PyBytes_AsStringAndSize()*는 `-1`을 반환하고 `TypeError`를 발생시킵니다.

[1] 정수 지정자 (d, u, ld, lu, zd, zu, i, x)에서: 0-변환 플래그는 정밀도를 지정해도 영향을 미칩니다.

버전 3.5에서 변경: 이전에는, 바이트열 객체에 널 바이트가 포함되어 있으면 TypeError가 발생했습니다.

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)

Part of the 안정 ABI. *bytes*에 *newpart*의 내용을 덧붙인 새 바이트열 객체를 **bytes*에 만듭니다; 호출자가 새 참조를 소유합니다. *bytes*의 예전 값에 대한 참조를 훔칩니다. 새 객체가 만들어질 수 없으면, *bytes*에 대한 예전 참조는 여전히 버려지고 **bytes*의 값은 NULL로 설정됩니다; 적절한 예외가 설정됩니다.

void **PyBytes_ConcatAndDel**(*PyObject* **bytes, *PyObject* *newpart)

Part of the 안정 ABI. *bytes*에 *newpart*의 내용을 덧붙인 새 바이트열 객체를 **bytes*에 만듭니다. 이 버전은 *newpart*로의 강한 참조를 해제합니다 (즉 참조 횟수를 감소시킵니다).

PyObject ***PyBytes_Join**(*PyObject* *sep, *PyObject* *iterable)

Similar to `sep.join(iterable)` in Python.

sep must be Python `bytes` object. (Note that *PyUnicode_Join()* accepts `NULL` separator and treats it as a space, whereas *PyBytes_Join()* doesn' t accept `NULL` separator.)

iterable must be an iterable object yielding objects that implement the *buffer protocol*.

On success, return a new `bytes` object. On error, set an exception and return `NULL`.

Added in version 3.14.

int **_PyBytes_Resize**(*PyObject* **bytes, *Py_ssize_t* newsize)

바이트열 객체의 크기를 변경합니다. *newsize*는 바이트열 객체의 새 길이가 됩니다. 새 바이트열 객체를 생성하고 이전 객체를 파괴한다고 생각할 수 있습니다. 단지 더 효율적으로 수행합니다. 기존 바이트열 객체의 주소를 lvalue(내용을 기록할 수 있습니다)로 전달하고, 원하는 새 크기를 전달합니다. 성공하면, **bytes*는 크기가 변경된 바이트열 객체를 갖게 되고 `0`이 반환됩니다; **bytes*의 주소는 입력값과 다를 수 있습니다. 재할당이 실패하면, **bytes*에 있는 원래 바이트열 객체는 할당 해제되고, **bytes*가 `NULL`로 설정되고, `MemoryError`가 설정되며 `-1`이 반환됩니다.

9.3.2 바이트 배열 객체

type **PyByteArrayObject**

이 *PyObject*의 서브 형은 파이썬 bytearray 객체를 나타냅니다.

PyTypeObject **PyByteArray_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 bytearray 형을 나타냅니다; 파이썬 계층의 `bytearray`와 같은 객체입니다.

형 검사 매크로

int **PyByteArray_Check**(*PyObject* *o)

객체 *o*가 bytearray 객체이거나 bytearray 형의 서브 형 인스턴스면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyByteArray_CheckExact**(*PyObject* *o)

객체 *o*가 bytearray 객체이지만, bytearray 형의 서브 형 인스턴스는 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

직접 API 함수

PyObject ***PyByteArray_FromObject**(*PyObject* *o)

반환값: 새 참조. *Part of the* 안정 ABI. 버퍼 프로토콜을 구현하는 임의의 객체(*o*)로부터 써서 새로운 bytearray 객체를 돌려줍니다.

실패하면, `NULL`을 반환하고 예외를 설정합니다.

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

반환값: 새 참조. *Part of the* 안정 ABI. *string*과 그 길이(*len*)로 새로운 bytearray 객체를 만듭니다.

실패하면, `NULL`을 반환하고 예외를 설정합니다.

PyObject ***PyByteArray_Concat**(*PyObject* *a, *PyObject* *b)

반환값: 새 참조. *Part of the* 안정 ABI. 바이트 배열 *a* 와 *b*를 이어붙여 새로운 bytearray로 반환합니다.

실패하면, `NULL`을 반환하고 예외를 설정합니다.

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

Part of the 안정 ABI. `NULL` 포인터를 확인한 후 *bytearray*의 크기를 반환합니다.

char ***PyByteArray_AsString**(*PyObject* *bytearray)

Part of the 안정 ABI. `NULL` 포인터를 확인한 후 *bytearray*의 내용을 char 배열로 반환합니다. 반환되는 배열에는 항상 여분의 널 바이트가 추가됩니다.

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

Part of the 안정 ABI. Resize the internal buffer of *bytearray* to *len*. Failure is a `-1` return with an exception set.

버전 3.14에서 변경: A negative *len* will now result in an exception being set and -1 returned.

매크로

이 매크로는 속도를 위해 안전을 희생하며 포인터를 확인하지 않습니다.

char ***PyByteArray_AS_STRING**(*PyObject* *bytearray)

*PyByteArray_AsString()*과 유사하지만 에러를 검사하지 않습니다.

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)

*PyByteArray_Size()*와 유사하지만 에러를 검사하지 않습니다.

9.3.3 유니코드 객체와 코덱

유니코드 객체

파이썬 3.3에서 **PEP 393**을 구현한 이후, 유니코드 객체는 내부적으로 다양한 표현을 사용하여 전체 유니코드 문자 범위를 처리하면서 메모리 효율성을 유지합니다. 모든 코드 포인트가 128, 256 또는 65536 미만인 문자열에 대한 특별한 경우가 있습니다; 그렇지 않으면, 코드 포인트는 1114112 (전체 유니코드 범위) 미만이어야 합니다.

UTF-8 representation is created on demand and cached in the Unicode object.

참고

Py_UNICODE 표현은 폐지된 API와 함께 파이썬 3.12에서 제거되었습니다. 자세한 정보는 **PEP 623**을 참조하십시오.

유니코드 형

다음은 파이썬에서 유니코드 구현에 사용되는 기본 유니코드 객체 형입니다:

PyTypeObject **PyUnicode_Type**

Part of the 안정 ABI. This instance of *PyTypeObject* represents the Python Unicode type. It is exposed to Python code as `str`.

PyTypeObject **PyUnicodeIter_Type**

Part of the 안정 ABI. This instance of *PyTypeObject* represents the Python Unicode iterator type. It is used to iterate over Unicode string objects.

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**

Part of the 안정 ABI. 이 형들은 각각 32비트, 16비트 및 8비트의 문자를 포함하기에 충분한 부호 없는 정수 형을 위한 typedef 입니다. 단일 유니코드 문자를 처리할 때는, *Py_UCS4*를 사용하십시오.

Added in version 3.3.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

이 *PyObject* 서브 형들은 파이썬 유니코드 객체를 나타냅니다. 거의 모든 경우에, 유니코드 객체를 처리하는 모든 API 함수가 *PyObject* 포인터를 취하고 반환하므로 직접 사용해서는 안 됩니다.

Added in version 3.3.

다음 API는 빠른 검사를 수행하고 유니코드 객체의 내부 읽기 전용 데이터에 액세스하는 데 사용할 수 있는 C 매크로와 정적 인라인 함수입니다:

int **PyUnicode_Check**(*PyObject* *obj)

객체 *obj*가 유니코드 객체이거나 유니코드 서브 형의 인스턴스이면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyUnicode_CheckExact**(*PyObject* *obj)

객체 *obj*가 유니코드 객체이지만, 서브 형의 인스턴스가 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

유니코드 문자열의 길이를 코드 포인트로 반환합니다. *unicode*는 "규범적(canonical)" 표현의 유니코드 객체여야 합니다 (검사하지 않습니다).

Added in version 3.3.

Py_UCS1 ***PyUnicode_1BYTE_DATA**(*PyObject* *unicode)

Py_UCS2 ***PyUnicode_2BYTE_DATA**(*PyObject* *unicode)

Py_UCS4 ***PyUnicode_4BYTE_DATA**(*PyObject* *unicode)

직접 문자 액세스를 위해 UCS1, UCS2 또는 UCS4 정수 형으로 캐스트 된 규범적(canonical) 표현에 대한 포인터를 반환합니다. 규범적(canonical) 표현이 올바른 문자 크기인지 검사하지 않습니다; *PyUnicode_KIND()*를 사용하여 올바른 함수를 선택하십시오.

Added in version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

PyUnicode_KIND() 매크로의 값을 반환합니다.

Added in version 3.3.

버전 3.12에서 변경: `PyUnicode_WCHAR_KIND`는 제거되었습니다.

int **PyUnicode_KIND**(*PyObject* *unicode)

이 유니코드 객체가 데이터를 저장하는 데 사용하는 문자 당 바이트 수를 나타내는 PyUnicode 종류 상수 (위를 참조하십시오) 중 하나를 반환합니다. *unicode*는 "규범적(canonical)" 표현의 유니코드 객체여야 합니다 (검사하지 않습니다).

Added in version 3.3.

void ***PyUnicode_DATA**(*PyObject* *unicode)

원시 유니코드 버퍼에 대한 void 포인터를 반환합니다. *unicode*는 "규범적(canonical)" 표현의 유니코드 객체여야 합니다 (검사하지 않습니다).

Added in version 3.3.

void **PyUnicode_WRITE**(int kind, void *data, *Py_ssize_t* index, *Py_UCS4* value)

Write the code point *value* to the given zero-based *index* in a string.

The *kind* value and *data* pointer must have been obtained from a string using *PyUnicode_KIND()* and *PyUnicode_DATA()* respectively. You must hold a reference to that string while calling PyUnicode_WRITE(). All requirements of *PyUnicode_WriteChar()* also apply.

The function performs no checks for any of its requirements, and is intended for usage in loops.

Added in version 3.3.

Py_UCS4 **PyUnicode_READ**(int kind, void *data, *Py_ssize_t* index)

규범적(canonical) 표현 *data*(*PyUnicode_DATA()*로 얻은 대로)에서 코드 포인트를 읽습니다. 검사나 준비(ready) 호출이 수행되지 않습니다.

Added in version 3.3.

Py_UCS4 **PyUnicode_READ_CHAR**(*PyObject* *unicode, *Py_ssize_t* index)

"규범적(canonical)" 표현이어야 하는, 유니코드 객체 *unicode*에서 문자를 읽습니다. 여러 연속 읽기를 수행한다면 *PyUnicode_READ()*보다 효율성이 떨어집니다.

Added in version 3.3.

Py_UCS4 **PyUnicode_MAX_CHAR_VALUE**(*PyObject* *unicode)

"규범적(canonical)" 표현이어야 하는, *unicode*를 기반으로 다른 문자열을 만드는 데 적합한 최대 코드 포인트를 반환합니다. 이것은 항상 근사치이지만 문자열을 이터레이트 하는 것보다 효율적입니다.

Added in version 3.3.

int **PyUnicode_IsIdentifier**(*PyObject* *unicode)

Part of the 안정 ABI. 언어 정의에 따라 문자열이 유효한 식별자이면 1을 반환합니다, 섹션 identifiers. 그렇지 않으면 0을 반환합니다.

버전 3.9에서 변경: 문자열이 준비(ready)되지 않았을 때, 이 함수는 더는 *Py_FatalError()*를 호출하지 않습니다.

unsigned int **PyUnicode_IS_ASCII**(*PyObject* *unicode)

Return true if the string only contains ASCII characters. Equivalent to str.isascii().

Added in version 3.2.

유니코드 문자 속성

유니코드는 다양한 문자 속성을 제공합니다. 가장 자주 필요한 것은 파이썬 구성에 따라 C 함수에 매핑되는 이러한 매크로를 통해 사용할 수 있습니다.

int **Py_UNICODE_ISSPACE**(*Py_UCS4* ch)

*ch*가 공백 문자인지에 따라 1이나 0을 반환합니다.

int **Py_UNICODE_ISLOWER**(*Py_UCS4* ch)

*ch*가 소문자인지에 따라 1이나 0을 반환합니다.

int **Py_UNICODE_ISUPPER**(*Py_UCS4* ch)

*ch*가 대문자인지에 따라 1이나 0을 반환합니다.

int **Py_UNICODE_ISTITLE**(*Py_UCS4* ch)

*ch*가 제목 케이스 문자인지에 따라 1이나 0을 반환합니다.

int **Py_UNICODE_ISLINEBREAK**(*Py_UCS4* ch)

*ch*가 줄 바꿈 문자인지에 따라 1이나 0을 반환합니다.

int **Py_UNICODE_ISDECIMAL**(*Py_UCS4* ch)

*ch*가 10진수 문자인지에 따라 1이나 0을 반환합니다.

int **Py_UNICODE_ISDIGIT**(*Py_UCS4* ch)
*ch*가 디짓(digit) 문자인지에 따라 `1`이나 `0`을 반환합니다.

int **Py_UNICODE_ISNUMERIC**(*Py_UCS4* ch)
*ch*가 숫자(numeric) 문자인지에 따라 `1`이나 `0`을 반환합니다.

int **Py_UNICODE_ISALPHA**(*Py_UCS4* ch)
*ch*가 알파벳 문자인지에 따라 `1`이나 `0`을 반환합니다.

int **Py_UNICODE_ISALNUM**(*Py_UCS4* ch)
*ch*가 영숫자 문자인지에 따라 `1`이나 `0`을 반환합니다.

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4* ch)
*ch*가 `str.isprintable()`의 의미에서 인쇄 가능한 문자인지에 따라 `1`이나 `0`을 반환합니다.

다음 API는 빠른 직접 문자 변환에 사용할 수 있습니다:

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4* ch)
소문자로 변환된 문자 *ch*를 반환합니다.

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)
대문자로 변환된 문자 *ch*를 반환합니다.

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)
제목 케이스로 변환된 문자 *ch*를 반환합니다.

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)
10진 양의 정수로 변환된 문자 *ch*를 반환합니다. 이것이 불가능하면 `-1`을 반환합니다. 이 함수는 예외를 발생시키지 않습니다.

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)
한 자리 정수로 변환된 문자 *ch*를 반환합니다. 이것이 불가능하면 `-1`을 반환합니다. 이 함수는 예외를 발생시키지 않습니다.

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)
double로 변환된 문자 *ch*를 반환합니다. 이것이 불가능하면 `-1.0`을 반환합니다. 이 함수는 예외를 발생시키지 않습니다.

다음 API를 사용하여 서로게이트를 다룰 수 있습니다:

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)
*ch*가 서로게이트인지 확인합니다 (`0xD800 <= ch <= 0xDFFF`).

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)
*ch*가 상위 서로게이트인지 확인합니다 (`0xD800 <= ch <= 0xDBFF`).

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)
*ch*가 하위 서로게이트인지 확인합니다 (`0xDC00 <= ch <= 0xDFFF`).

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)
두 서로게이트 코드 포인트를 결합하고 단일 *Py_UCS4* 값을 반환합니다. *high*와 *low*는 각각 서로게이트 쌍의 선행과 후행 서로게이트입니다. *high*는 [0xD800; 0xDBFF] 범위 내에 있어야 하며, *low*는 [0xDC00; 0xDFFF] 범위 내에 있어야 합니다.

유니코드 문자열 생성과 액세스

유니코드 객체를 만들고 기본 시퀀스 속성에 액세스하려면 다음 API를 사용하십시오:

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)
반환값: 새 참조. 새 유니코드 객체를 만듭니다. *maxchar*은 문자열에 배치할 실제 최대 코드 포인트여야 합니다. 근삿값으로, 127, 255, 65535, 1114111 시퀀스에서 가장 가까운 값으로 올림 할 수 있습니다.

On error, set an exception and return NULL.

After creation, the string can be filled by *PyUnicode_WriteChar()*, *PyUnicode_CopyCharacters()*, *PyUnicode_Fill()*, *PyUnicode_WRITE()* or similar. Since strings are supposed to be immutable, take care to not "use" the result while it is being modified. In particular, before it' s filled with its final contents, a string:

- must not be hashed,
- must not be *converted to UTF-8*, or another non-"canonical" representation,
- must not have its reference count changed,
- must not be shared with code that might do one of the above.

This list is not exhaustive. Avoiding these uses is your responsibility; Python does not always check these requirements.

To avoid accidentally exposing a partially-written string object, prefer using the *PyUnicodeWriter* API, or one of the PyUnicode_From* functions below.

Added in version 3.3.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

반환값: 새 참조. 주어진 *kind*(가능한 값은 *PyUnicode_KIND()*에 의해 반환된 *PyUnicode_1BYTE_KIND* 등입니다)로 새로운 유니코드 객체를 만듭니다. *buffer*는 kind에 따라 문자 당 1, 2 또는 4바이트의 *size* 단위의 배열을 가리켜야 합니다.

If necessary, the input *buffer* is copied and transformed into the canonical representation. For example, if the *buffer* is a UCS4 string (*PyUnicode_4BYTE_KIND*) and it consists only of codepoints in the UCS1 range, it will be transformed into UCS1 (*PyUnicode_1BYTE_KIND*).

Added in version 3.3.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

반환값: 새 참조. *Part of the* 안정 ABI. char 버퍼 *str*에서 유니코드 객체를 만듭니다. 바이트는 UTF-8로 인코딩된 것으로 해석됩니다. 버퍼는 새 객체에 복사됩니다. 반환 값은 공유 객체일 수 있습니다, 즉, 데이터 수정이 허용되지 않습니다.

이 함수는 다음과 같은 경우에 SystemError를 발생시킵니다:

- *size* < 0,
- *str* is NULL and *size* > 0

버전 3.12에서 변경: *str* == NULL with *size* > 0 is not allowed anymore.

PyObject ***PyUnicode_FromString**(const char *str)

반환값: 새 참조. *Part of the* 안정 ABI. UTF-8로 인코딩된 널-종료 char 버퍼 *str*에서 유니코드 객체를 만듭니다.

PyObject ***PyUnicode_FromFormat**(const char *format, ...)

반환값: 새 참조. *Part of the* 안정 ABI. C printf()-스타일 *format* 문자열과 가변 개수의 인자를 취해서, 결과 파이썬 유니코드 문자열의 크기를 계산하고 포맷된 값이 들어간 문자열을 반환합니다. 변수 인자는 C형이어야 하며 *format* ASCII 인코딩된 문자열의 포맷 문자와 정확히 일치해야 합니다.

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The '%' character, which marks the start of the specifier.
2. Conversion flags (optional), which affect the result of some conversion types.
3. Minimum field width (optional). If specified as an '*' (asterisk), the actual width is given in the next argument, which must be of type int, and the object to convert comes after the minimum field width and optional precision.

4. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is given in the next argument, which must be of type `int`, and the value to convert comes after the precision.
5. Length modifier (optional).
6. Conversion type.

The conversion flag characters are:

Flag	Meaning
`0`	The conversion will be zero padded for numeric values.
`-`	The converted value is left adjusted (overrides the `0` flag if both are given).

The length modifiers for following integer conversions (`d`, `i`, `o`, `u`, `x`, or `X`) specify the type of the argument (`int` by default):

Modifier	형
`l`	`long` 또는 `unsigned long`
`ll`	`long long` 또는 `unsigned long long`
`j`	`intmax_t` 또는 `uintmax_t`
`z`	`size_t` 또는 `ssize_t`
`t`	`ptrdiff_t`

The length modifier `l` for following conversions `s` or `V` specify that the type of the argument is `const wchar_t*`.

The conversion specifiers are:

Con-version Speci-fier	형	주석
`%`	*n/a*	리터럴 `%` 문자.
`d`, `i`	Specified by the length modifier	The decimal representation of a signed C integer.
`u`	Specified by the length modifier	The decimal representation of an unsigned C integer.
`o`	Specified by the length modifier	The octal representation of an unsigned C integer.
`x`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (lowercase).
`X`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (uppercase).
`c`	`int`	단일 문자.
`s`	`const char*` 또는 `const wchar_t*`	널-종료 C 문자 배열.
`p`	`const void*`	C 포인터의 16진수 표현. 플랫폼의 `printf`가 산출하는 내용과 관계없이 리터럴 `0x`로 시작하는 것이 보장된다는 점을 제외하면 거의 `printf("%p")`와 동등합니다.
`A`	*PyObject**	`ascii()`를 호출한 결과.
`U`	*PyObject**	유니코드 객체.
`V`	*PyObject**, `const char*` 또는 `const wchar_t*`	유니코드 객체(`NULL`일 수 있습니다)와 두 번째 매개 변수로서 널-종료 C 문자 배열 (첫 번째 매개 변수가 `NULL`이면 사용됩니다).
`S`	*PyObject**	*PyObject_Str()*을 호출한 결과.
`R`	*PyObject**	*PyObject_Repr()*을 호출한 결과.
`T`	*PyObject**	Get the fully qualified name of an object type; call *PyType_GetFullyQualifiedName()*.
`#T`	*PyObject**	Similar to `T` format, but use a colon (`:`) as separator between the module name and the qualified name.
`N`	*PyTypeObject**	Get the fully qualified name of a type; call *PyType_GetFullyQualifiedName()*.
`#N`	*PyTypeObject**	Similar to `N` format, but use a colon (`:`) as separator between the module name and the qualified name.

참고

너비 포매터 단위는 바이트가 아닌 문자 수입니다. 정밀도 포매터 단위는 `"%s"`와 `"%V"`의 경우는 바이트 수나 (길이 수정자 `l`이 사용되면) `wchar_t` 항목이고 (`PyObject*` 인자가 `NULL`이면), `"%A"`, `"%U"`, `"%S"`, `"%R"` 및 `"%V"`의 경우 문자 수입니다 (`PyObject*` 인자가 `NULL`이 아니면).

참고

Unlike to C `printf()` the `0` flag has effect even when a precision is given for integer conversions (`d`, `i`, `u`, `o`, `x`, or `X`).

버전 3.2에서 변경: `"%lld"`와 `"%llu"`에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: `"%li"`, `"%lli"` 및 `"%zi"`에 대한 지원이 추가되었습니다.

버전 3.4에서 변경: `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"`에 대한 너비와 정밀도 포매터 지원이 추가되었습니다.

버전 3.12에서 변경: Support for conversion specifiers `o` and `X`. Support for length modifiers `j` and `t`. Length modifiers are now applied to all integer conversions. Length modifier `l` is now applied to conversion specifiers `s` and `V`. Support for variable width and precision `*`. Support for flag `-`.

인식할 수 없는 포맷 문자는 이제 `SystemError`를 설정합니다. 이전 버전에서는 나머지 포맷 문자열이 모두 결과 문자열에 그대로 복사되고, 추가 인자는 버려지도록 했습니다.

버전 3.13에서 변경: `%T`, `%#T`, `%N` 및 `%#N`에 대한 지원이 추가되었습니다.

PyObject ***PyUnicode_FromFormatV**(const char *format, va_list vargs)

반환값: 새 참조. *Part of the* 안정 ABI. 정확히 두 개의 인자를 취한다는 점을 제외하면 *PyUnicode_FromFormat()*과 동일합니다.

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

반환값: 새 참조. *Part of the* 안정 ABI. 필요하면 유니코드 서브 형의 인스턴스를 새로운 진짜 유니코드 객체에 복사합니다. *obj*가 이미 (서브 형이 아닌) 진짜 유니코드 객체이면, 객체에 대한 새로운 강한 참조를 반환합니다.

유니코드나 이의 서브 형 이외의 객체는 `TypeError`를 발생시킵니다.

PyObject ***PyUnicode_FromOrdinal**(int ordinal)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 유니코드 코드 포인트 *ordinal*로 유니코드 객체를 만듭니다.

The ordinal must be in `range(0x110000)`. A `ValueError` is raised in the case it is not.

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 인코딩된 객체 *obj*를 유니코드 객체로 디코딩합니다.

`bytes`, `bytearray` 및 기타 바이트열류 객체는 주어진 *encoding*에 따라 *errors*로 정의한 에러 처리를 사용하여 디코딩됩니다. 둘 다 `NULL`이 될 수 있고, 이 경우 인터페이스는 기본값을 사용합니다 (자세한 내용은 내장 코덱를 참조하십시오).

유니코드 객체를 포함한 다른 모든 객체는 `TypeError`가 설정되도록 합니다.

API는 에러가 있으면 `NULL`을 반환합니다. 호출자는 반환된 객체의 참조 횟수를 감소시킬 책임이 있습니다.

void **PyUnicode_Append**(*PyObject* **p_left, *PyObject* *right)

Part of the 안정 ABI. Append the string *right* to the end of *p_left*. *p_left* must point to a *strong reference* to a Unicode object; `PyUnicode_Append()` releases ("steals") this reference.

On error, set **p_left* to `NULL` and set an exception.

On success, set **p_left* to a new strong reference to the result.

void **PyUnicode_AppendAndDel**(*PyObject* **p_left, *PyObject* *right)

Part of the 안정 ABI. The function is similar to *PyUnicode_Append()*, with the only difference being that it decrements the reference count of *right* by one.

PyObject ***PyUnicode_BuildEncodingMap**(*PyObject* *string)

반환값: 새 참조. *Part of the* 안정 ABI. Return a mapping suitable for decoding a custom single-byte encoding. Given a Unicode string *string* of up to 256 characters representing an encoding table, returns either a compact internal mapping object or a dictionary mapping character ordinals to byte values. Raises a `TypeError` and return `NULL` on invalid input.

Added in version 3.2.

const char ***PyUnicode_GetDefaultEncoding**(void)

Part of the 안정 ABI. Return the name of the default string encoding, `"utf-8"`. See `sys.getdefaultencoding()`.

The returned string does not need to be freed, and is valid until interpreter shutdown.

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

Part of the 안정 ABI 버전 *3.7* 이후로. 유니코드 객체의 길이를 코드 포인트로 반환합니다.

On error, set an exception and return `-1`.

Added in version 3.3.

Py_ssize_t **PyUnicode_CopyCharacters**(*PyObject* *to, *Py_ssize_t* to_start, *PyObject* *from, *Py_ssize_t* from_start, *Py_ssize_t* how_many)

한 유니코드 객체에서 다른 객체로 문자를 복사합니다. 이 함수는 필요하면 문자 변환을 수행하고 가능하면 `memcpy()`로 폴백합니다. 에러 시 `-1`을 반환하고 예외를 설정합니다, 그렇지 않으면 복사된 문자 수를 반환합니다.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Added in version 3.3.

int **PyUnicode_Resize**(*PyObject* **unicode, *Py_ssize_t* length);

Part of the 안정 ABI. Resize a Unicode object **unicode* to the new *length* in code points.

Try to resize the string in place (which is usually faster than allocating a new string and copying characters), or create a new string.

**unicode* is modified to point to the new (resized) object and `0` is returned on success. Otherwise, `-1` is returned and an exception is set, and **unicode* is left untouched.

The function doesn' t check string content, the result may not be a string in canonical representation.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

문자로 문자열을 채웁니다: *fill_char*을 `unicode[start:start+length]`에 씁니다.

*fill_char*이 문자열 최대 문자보다 크거나, 문자열에 둘 이상의 참조가 있으면 실패합니다.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

기록된 문자 수를 반환하거나, 에러 시 `-1`을 반환하고 예외를 발생시킵니다.

Added in version 3.3.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

Part of the 안정 ABI 버전 *3.7* 이후로. Write a *character* to the string *unicode* at the zero-based *index*. Return `0` on success, `-1` on error with an exception set.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object' s reference count is one). See *PyUnicode_WRITE()* for a version that skips these checks, making them your responsibility.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Added in version 3.3.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

Part of the 안정 ABI 버전 *3.7* 이후로. 문자열에서 문자를 읽습니다. 이 함수는 에러를 검사하지 않는 *PyUnicode_READ_CHAR()*와 달리 *unicode*가 유니코드 객체이고 인덱스가 범위를 벗어났는지 확인합니다.

성공 시 문자를, 에러 시 예외를 설정하고 `-1`을 반환합니다.

Added in version 3.3.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. 문자 인덱스 *start*(포함합니다)에서 문자 인덱스 *end*(제외합니다)까지 *unicode*의 하위 문자열을 반환합니다. 음수 인덱스는 지원되지 않습니다. 에러 시, 예외를 설정하고 `NULL`을 반환합니다.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

Part of the 안정 ABI 버전 *3.7* 이후로. *copy_null*이 설정되면, 널 문자를 포함하여 문자열 *unicode*를 UCS4 버퍼에 복사합니다. 에러 시 `NULL`을 반환하고 예외를 설정합니다 (특히, *buflen*이 *unicode*의 길이보다 작으면 `SystemError`). 성공하면 *buffer*가 반환됩니다.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

Part of the 안정 ABI 버전 *3.7* 이후로. 문자열 *unicode*를 *PyMem_Malloc()*을 사용하여 할당된 새 UCS4 버퍼에 복사합니다. 이것이 실패하면, `NULL`이 반환되고 `MemoryError`가 설정됩니다. 반환된 버퍼에는 항상 추가 널 코드 포인트가 있습니다.

Added in version 3.3.

로케일 인코딩

현재 로케일 인코딩을 사용하여 운영 체제에서 온 텍스트를 디코딩 할 수 있습니다.

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. 안드로이드와 VxWorks의 UTF-8이나 다른 플랫폼의 현재 로케일 인코딩의 문자열을 디코딩합니다. 지원되는 에러 처리기는 `"strict"`와 `"surrogateescape"`(**PEP 383**)입니다. 디코더는 *errors*가 `NULL`이면 `"strict"` 에러 처리기를 사용합니다. *str*은 널 문자로 끝나야 하지만 널 문자를 포함할 수 없습니다.

파일 시스템 인코딩과 에러 처리기로 문자열을 디코딩하려면 *PyUnicode_DecodeFSDefaultAndSize()*를 사용하십시오.

이 함수는 파이썬 UTF-8 모드를 무시합니다.

더 보기

Py_DecodeLocale() 함수.

Added in version 3.3.

버전 3.7에서 변경: 이 함수는 이제 안드로이드를 제외하고 `surrogateescape` 에러 처리기에 현재 로케일 인코딩도 사용합니다. 이전에는, *Py_DecodeLocale()*이 `surrogateescape`에 사용되었고, 현재 로케일 인코딩은 `strict`에 사용되었습니다.

PyObject ***PyUnicode_DecodeLocale**(const char *str, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. *PyUnicode_DecodeLocaleAndSize()*와 유사하지만, `strlen()`을 사용하여 문자열 길이를 계산합니다.

Added in version 3.3.

PyObject ***PyUnicode_EncodeLocale**(*PyObject* *unicode, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. 유니코드 객체를 안드로이드와 VxWorks에서 UTF-8로 인코딩하거나, 다른 플랫폼에서 현재 로케일 인코딩으로 인코딩합니다. 지원되는 에러 처리기는 `"strict"`와 `"surrogateescape"`(**PEP 383**)입니다. 인코더는 *errors*가 `NULL`이면 `"strict"` 에러 처리기를 사용합니다. `bytes` 객체를 반환합니다. *unicode*는 내장된 널 문자를 포함할 수 없습니다.

파일시스템 인코딩과 에러 처리기로 문자열을 인코딩하려면 *PyUnicode_EncodeFSDefault()*를 사용하십시오.

이 함수는 파이썬 UTF-8 모드를 무시합니다.

더 보기

Py_EncodeLocale() 함수.

Added in version 3.3.

버전 3.7에서 변경: 이 함수는 이제 안드로이드를 제외하고 `surrogateescape` 에러 처리기에 현재 로케일 인코딩도 사용합니다. 이전에는 *Py_EncodeLocale()*이 `surrogateescape`에 사용되었고, 현재 로케일 인코딩은 `strict`에 사용되었습니다.

파일 시스템 인코딩

Functions encoding to and decoding from the *filesystem encoding and error handler* (**PEP 383** and **PEP 529**).

인자 구문 분석 중에 파일 이름을 `bytes`로 인코딩하려면, `"O&"` 변환기를 사용하고 `PyUnicode_FSConverter()`를 변환 함수로 전달해야 합니다:

int **PyUnicode_FSConverter**(*PyObject* *obj, void *result)

Part of the 안정 ABI. *PyArg_Parse** 변환기: (직접 또는 `os.PathLike` 인터페이스를 통해 얻은) `str` 객체를 *PyUnicode_EncodeFSDefault()*를 사용하여 `bytes`로 인코딩합니다; `bytes` 객체는 있는 그대로의 출력입니다. *result*는 형이 *PyObject** (또는 *PyBytesObject**) 인 C 변수의 주소여야 합니다. 성공 시, 더는 사용되지 않을 때 해제해야 하는 바이트열 객체에 대한 강한 참조로 변수를 설정하고 0이 아닌 값(*Py_CLEANUP_SUPPORTED*)을 반환합니다. 결과에 내장된 널 바이트는 허용되지 않습니다. 실패 시, 예외를 설정하고 `0`을 반환합니다.

If *obj* is `NULL`, the function releases a strong reference stored in the variable referred by *result* and returns `1`.

Added in version 3.1.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

인자 구문 분석 중에 파일 이름을 `str`로 디코딩하려면, `"O&"` 변환기를 사용하고 `PyUnicode_FSDecoder()`를 변환 함수로 전달해야 합니다:

int **PyUnicode_FSDecoder**(*PyObject* *obj, void *result)

Part of the 안정 ABI. *PyArg_Parse** 변환기: (직접 또는 `os.PathLike` 인터페이스를 통해 간접적으로 얻은) `bytes` 객체를 *PyUnicode_DecodeFSDefaultAndSize()*를 사용하여 `str`로 디코딩합니다; `str` 객체는 있는 그대로의 출력입니다. *result*는 형이 *PyObject** (또는 *PyUnicodeObject**) 인 C 변수의 주소여야 합니다. 성공 시, 더는 사용되지 않을 때 해제해야 하는 유니코드 객체에 대한 강한 참조로 변수를 설정하고 0이 아닌 값(*Py_CLEANUP_SUPPORTED*)을 반환합니다. 결과에 내장된 널 바이트는 허용되지 않습니다. 실패 시, 예외를 설정하고 `0`을 반환합니다.

If *obj* is `NULL`, release the strong reference to the object referred to by *result* and return `1`.

Added in version 3.2.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

PyObject ***PyUnicode_DecodeFSDefaultAndSize**(const char *str, *Py_ssize_t* size)

반환값: 새 참조. *Part of the* 안정 ABI. Decode a string from the *filesystem encoding and error handler*.

현재 로케일 인코딩에서 문자열을 디코딩해야 하면, *PyUnicode_DecodeLocaleAndSize()*를 사용하십시오.

더 보기

Py_DecodeLocale() 함수.

버전 3.6에서 변경: The *filesystem error handler* is now used.

PyObject ***PyUnicode_DecodeFSDefault**(const char *str)

반환값: 새 참조. *Part of the* 안정 ABI. 파일시스템 인코딩과 에러 처리기로 널 종료 문자열을 디코딩합니다.

문자열 길이가 알려져 있으면, *PyUnicode_DecodeFSDefaultAndSize()*를 사용하십시오.

버전 3.6에서 변경: The *filesystem error handler* is now used.

PyObject ***PyUnicode_EncodeFSDefault**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. 유니코드 객체를 파일시스템 인코딩과 에러 처리기로 인코딩하고, `bytes`를 반환합니다. 결과 `bytes` 객체에는 널 바이트가 포함될 수 있음에 유의하십시오.

현재 로케일 인코딩으로 문자열을 인코딩해야 하면, *PyUnicode_EncodeLocale()*을 사용하십시오.

> **더 보기**
>
> *Py_EncodeLocale()* 함수.

Added in version 3.2.

버전 3.6에서 변경: The *filesystem error handler* is now used.

wchar_t 지원

지원하는 플랫폼에 대한 `wchar_t` 지원:

PyObject ***PyUnicode_FromWideChar**(const wchar_t *wstr, *Py_ssize_t* size)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 *size*의 `wchar_t` 버퍼 *wstr*에서 유니코드 객체를 만듭니다. `-1`을 *size*로 전달하면 함수가 `wcslen()`을 사용하여 길이를 스스로 계산해야 함을 나타냅니다. 실패하면 `NULL`을 반환합니다.

Py_ssize_t ***PyUnicode_AsWideChar**(*PyObject* *unicode, wchar_t *wstr, *Py_ssize_t* size)

Part of the 안정 ABI. 유니코드 객체 내용을 `wchar_t` 버퍼 *wstr*에 복사합니다. 최대 *size* `wchar_t` 문자가 복사됩니다 (후행 널 종료 문자가 제외될 수 있습니다). 복사된 `wchar_t` 문자 수나 에러가 발생하면 `-1`을 반환합니다.

When *wstr* is `NULL`, instead return the *size* that would be required to store all of *unicode* including a terminating null.

결과 `wchar_t*` 문자열은 널로 종료될 수도 있고 아닐 수도 있음에 유의하십시오. 응용 프로그램에 필요하면 `wchar_t*` 문자열이 널로 끝나는지 확인하는 것은 호출자의 책임입니다. 또한, `wchar_t*` 문자열에는 널 문자가 포함될 수 있으며, 이로 인해 대부분의 C 함수와 함께 사용될 때 문자열이 잘리게 됨에 유의하십시오.

wchar_t ***PyUnicode_AsWideCharString**(*PyObject* *unicode, *Py_ssize_t* *size)

Part of the 안정 ABI 버전 *3.7* 이후로. 유니코드 객체를 와이드 문자 문자열로 변환합니다. 출력 문자열은 항상 널 문자로 끝납니다. *size*가 `NULL`이 아니면, (후행 널 종료 문자를 제외한) 와이드 문자의 수를 **size*에 씁니다. 결과 `wchar_t` 문자열이 널 문자를 포함할 수 있고, 이로 인해 대부분의 C 함수와 함께 사용될 때 문자열이 잘리게 됨에 유의하십시오. *size*가 `NULL`이고 `wchar_t*` 문자열이 널 문자를 포함하면 `ValueError`가 발생합니다.

성공 시 *PyMem_New*에 의해 할당된 버퍼를 반환합니다 (*PyMem_Free()*를 사용하여 해제하십시오). 에러 시, `NULL`을 반환하고 **size*는 정의되지 않습니다. 메모리 할당이 실패하면 `MemoryError`를 발생시킵니다.

Added in version 3.2.

버전 3.7에서 변경: *size*가 `NULL`이고 `wchar_t*` 문자열이 널 문자를 포함하면 `ValueError`를 발생시킵니다.

내장 코덱

파이썬은 속도를 위해 C로 작성된 내장 코덱 집합을 제공합니다. 이러한 코덱들은 모두 다음 함수들을 통해 직접 사용할 수 있습니다.

다음 API의 대부분은 두 개의 인자 encoding과 errors를 취하며, 내장 `str()` 문자열 객체 생성자의 것들과 같은 의미입니다.

encoding을 `NULL`로 설정하면 기본 인코딩인 UTF-8이 사용됩니다. 파일 시스템 호출은 파일 이름 인코딩에 *PyUnicode_FSConverter()*를 사용해야 합니다. 이것은 내부적으로 파일시스템 인코딩과 에러 처리기를 사용합니다.

에러 처리는 errors로 설정되는데, 코덱에 대해 정의된 기본 처리를 사용함을 의미하는 NULL로 설정될 수도 있습니다. 모든 내장 코덱에 대한 기본 에러 처리는 "strict" 입니다 (ValueError가 발생합니다).

코덱은 모두 유사한 인터페이스를 사용합니다. 단순성을 위해 다음에 나오는 일반 코덱과의 차이만 설명합니다.

일반 코덱

The following macro is provided:

Py_UNICODE_REPLACEMENT_CHARACTER

The Unicode code point U+FFFD (replacement character).

This Unicode character is used as the replacement character during decoding if the *errors* argument is set to "replace".

다음은 일반 코덱 API입니다:

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. *encoding*과 *errors*는 str() 내장 함수의 같은 이름의 매개 변수와 같은 의미입니다. 사용할 코덱은 파이썬 코덱 레지스트리를 사용하여 조회됩니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_AsEncodedString**(*PyObject* *unicode, const char *encoding, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. *encoding*과 *errors*는 유니코드 encode() 메서드의 같은 이름의 매개 변수와 같은 의미입니다. 사용할 코덱은 파이썬 코덱 레지스트리를 사용하여 조회됩니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

UTF-8 코덱

다음은 UTF-8 코덱 API입니다:

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. UTF-8로 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

반환값: 새 참조. *Part of the* 안정 ABI. *consumed*가 NULL이면, *PyUnicode_DecodeUTF8()*처럼 동작합니다. *consumed*가 NULL이 아니면, 후행 불완전한 UTF-8 바이트 시퀀스는 에러로 처리되지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 *consumed*에 저장됩니다.

PyObject ***PyUnicode_AsUTF8String**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. UTF-8을 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

The function fails if the string contains surrogate code points (U+D800 - U+DFFF).

const char ***PyUnicode_AsUTF8AndSize**(*PyObject* *unicode, *Py_ssize_t* *size)

Part of the 안정 ABI 버전 *3.10* 이후로. 유니코드 객체의 UTF-8 인코딩에 대한 포인터를 반환하고, 인코딩된 표현의 크기를 (바이트 단위로) *size*에 저장합니다. *size* 인자는 NULL일 수 있습니다; 이 경우 크기가 저장되지 않습니다. 반환된 버퍼에는 다른 널 코드 포인트가 있는지에 관계없이, 항상 추가 널 바이트가 추가됩니다 (*size*에 포함되지 않습니다).

On error, set an exception, set *size* to -1 (if it's not NULL) and return NULL.

The function fails if the string contains surrogate code points (U+D800 - U+DFFF).

이것은 유니코드 객체에서 문자열의 UTF-8 표현을 캐시하고, 후속 호출은 같은 버퍼에 대한 포인터를 반환합니다. 호출자는 버퍼 할당 해제에 대한 책임이 없습니다. 유니코드 객체가 가비지 수거될 때 버퍼는 할당 해제되고 버퍼를 가리키는 포인터는 유효하지 않게 됩니다.

Added in version 3.3.

버전 3.7에서 변경: 반환형은 이제 `char *`가 아니라 `const char *`입니다.

버전 3.10에서 변경: This function is a part of the *limited API*.

const char ***PyUnicode_AsUTF8**(*PyObject* *unicode)

*PyUnicode_AsUTF8AndSize()*와 같지만, 크기를 저장하지 않습니다.

> **경고**
>
> This function does not have any special behavior for null characters embedded within *unicode*. As a result, strings containing null characters will remain in the returned string, which some C functions might interpret as the end of the string, leading to truncation. If truncation is an issue, it is recommended to use *PyUnicode_AsUTF8AndSize()* instead.

Added in version 3.3.

버전 3.7에서 변경: 반환형은 이제 `char *`가 아니라 `const char *`입니다.

UTF-32 코덱

다음은 UTF-32 코덱 API입니다:

PyObject ***PyUnicode_DecodeUTF32**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

반환값: 새 참조. *Part of the* 안정 ABI. UTF-32로 인코딩된 버퍼 문자열에서 *size* 바이트를 디코딩하고 해당 유니코드 객체를 반환합니다. *errors*(`NULL`이 아니면)는 에러 처리를 정의합니다. 기본값은 "strict" 입니다.

*byteorder*가 `NULL`이 아니면, 디코더는 지정된 바이트 순서를 사용하여 디코딩을 시작합니다:

```
*byteorder == -1:  리틀 엔디안
*byteorder == 0:   네이티브 엔디안
*byteorder == 1:   빅 엔디안
```

`*byteorder`가 0이고, 입력 데이터의 처음 4바이트가 바이트 순서 표시(BOM)이면, 디코더가 이 바이트 순서로 전환되고 BOM은 결과 유니코드 문자열에 복사되지 않습니다. `*byteorder`가 `-1`이나 `1`이면, 모든 바이트 순서 표시가 출력에 복사됩니다.

완료 후, **byteorder*는 입력 데이터의 끝에서 현재 바이트 순서로 설정됩니다.

*byteorder*가 `NULL`이면, 코덱은 네이티브 순서 모드로 시작합니다.

코덱에서 예외가 발생하면 `NULL`을 반환합니다.

PyObject ***PyUnicode_DecodeUTF32Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

반환값: 새 참조. *Part of the* 안정 ABI. *consumed*가 `NULL`이면, *PyUnicode_DecodeUTF32()*처럼 동작합니다. *consumed*가 `NULL`이 아니면, *PyUnicode_DecodeUTF32Stateful()* 은 후행 불완전 UTF-32 바이트 시퀀스(가령 4로 나누어떨어지지 않는 바이트 수)를 에러로 처리하지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 *consumed*에 저장됩니다.

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. 네이티브 바이트 순서로 UTF-32 인코딩을 사용하여 파이썬 바이트 문자열을 반환합니다. 문자열은 항상 BOM 마크로 시작합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 `NULL`을 반환합니다.

UTF-16 코덱

다음은 UTF-16 코덱 API입니다:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

반환값: 새 참조. *Part of the* 안정 ABI. UTF-16으로 인코딩된 버퍼 문자열에서 *size* 바이트를 디코딩하고 해당 유니코드 객체를 반환합니다. *errors*(NULL이 아니면)는 에러 처리를 정의합니다. 기본값은 "strict" 입니다.

*byteorder*가 NULL이 아니면, 디코더는 지정된 바이트 순서를 사용하여 디코딩을 시작합니다:

```
*byteorder == -1:  리틀 엔디안
*byteorder == 0:   네이티브 엔디안
*byteorder == 1:   빅 엔디안
```

*byteorder*가 0이고, 입력 데이터의 처음 2바이트가 바이트 순서 표시(BOM)이면, 디코더는 이 바이트 순서로 전환되고 BOM은 결과 유니코드 문자열에 복사되지 않습니다. *byteorder*가 -1이나 1이면 모든 바이트 순서 표시가 출력에 복사됩니다(\ufeff나 \ufffe 문자가 됩니다).

완료 후, *byteorder*는 입력 데이터의 끝에서 현재 바이트 순서로 설정됩니다.

*byteorder*가 NULL이면, 코덱은 네이티브 순서 모드로 시작합니다.

코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_DecodeUTF16Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

반환값: 새 참조. *Part of the* 안정 ABI. *consumed*가 NULL이면, *PyUnicode_DecodeUTF16()*처럼 동작합니다. *consumed*가 NULL이 아니면, *PyUnicode_DecodeUTF16Stateful()* 은 후행 불완전 UTF-16 바이트 시퀀스(가령 홀수 바이트 수나 분할 서로게이트 쌍)를 에러로 처리하지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 *consumed*에 저장됩니다.

PyObject ***PyUnicode_AsUTF16String**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. 네이티브 바이트 순서로 UTF-16 인코딩을 사용하여 파이썬 바이트 문자열을 반환합니다. 문자열은 항상 BOM 마크로 시작합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

UTF-7 코덱

다음은 UTF-7 코덱 API입니다:

PyObject ***PyUnicode_DecodeUTF7**(const char *str, *Py_ssize_t* size, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. UTF-7로 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_DecodeUTF7Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

반환값: 새 참조. *Part of the* 안정 ABI. *consumed*가 NULL이면, *PyUnicode_DecodeUTF7()*처럼 동작합니다. *consumed*가 NULL이 아니면, 후행 불완전한 UTF-7 base-64 섹션은 에러로 처리되지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 *consumed*에 저장됩니다.

유니코드 이스케이프 코덱

다음은 "유니코드 이스케이프(Unicode Escape)" 코덱 API입니다:

PyObject ***PyUnicode_DecodeUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 유니코드 이스케이프 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_AsUnicodeEscapeString**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. 유니코드 이스케이프를 사용하여 유니코드 객체를 인코딩하고 결과를 bytes 객체로 반환합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

원시 유니코드 이스케이프 코덱

다음은 "원시 유니코드 이스케이프(Raw Unicode Escape)" 코덱 API입니다:

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 원시 유니코드 이스케이프 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. 원시 유니코드 이스케이프를 사용하여 유니코드 객체를 인코딩하고 결과를 bytes 객체로 반환합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Latin-1 코덱

다음은 Latin-1 코덱 API입니다: Latin-1은 처음 256개의 유니코드 서수에 해당하며 인코딩 중에 코덱에서 이들만 허용됩니다.

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. Latin-1 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. Latin-1을 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

ASCII 코덱

다음은 ASCII 코덱 API입니다. 7비트 ASCII 데이터만 허용됩니다. 다른 모든 코드는 에러를 생성합니다.

PyObject ***PyUnicode_DecodeASCII**(const char *str, *Py_ssize_t* size, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. ASCII 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI. ASCII를 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

문자 맵 코덱

이 코덱은 다양한 코덱을 구현하는 데 사용할 수 있다는 점에서 특별합니다 (실제로 encodings 패키지에 포함된 대부분의 표준 코덱을 얻기 위해 수행되었습니다). 코덱은 매핑을 사용하여 문자를 인코딩하고 디코딩합니다. 제공된 매핑 객체는 `__getitem__()` 매핑 인터페이스를 지원해야 합니다; 딕셔너리와 시퀀스가 잘 작동합니다.

다음은 매핑 코덱 API입니다:

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 *mapping* 객체를 사용하여 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

*mapping*이 NULL이면, Latin-1 디코딩이 적용됩니다. 그렇지 않으면 *mapping*은 바이트 서수(0에서 255 사이의 정수)를 유니코드 문자열, 정수(유니코드 서수로 해석됩니다) 또는 None으로 매핑해야합니다. 매핑되지 않은 데이터 바이트(None, 0xFFFE 또는 '\ufffe'로 매핑되는 것뿐만 아니라, LookupError를 유발하는 것)은 정의되지 않은 매핑으로 처리되어 에러를 발생시킵니다.

PyObject ***PyUnicode_AsCharmapString**(*PyObject* *unicode, *PyObject* *mapping)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 *mapping* 객체를 사용하여 유니코드 객체를 인코딩하고 결과를 bytes 객체로 반환합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

mapping 객체는 유니코드 서수 정수를 bytes 객체, 0에서 255 사이의 정수 또는 `None`으로 매핑해야 합니다. `None`에 매핑되는 것뿐만 아니라 매핑되지 않은 문자 서수(`LookupError`를 유발하는 것)는 "정의되지 않은 매핑"으로 처리되어 에러가 발생합니다.

다음 코덱 API는 유니코드를 유니코드로 매핑한다는 점에서 특별합니다.

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI. 문자 매핑 테이블을 적용하여 문자열을 변환하고 결과 유니코드 객체를 반환합니다. 코덱에서 예외가 발생하면 `NULL`을 반환합니다.

매핑 테이블은 유니코드 서수 정수를 유니코드 서수 정수나 `None`(문자가 삭제되도록 합니다)에 매핑해야 합니다.

매핑 테이블은 `__getitem__()` 인터페이스 만 제공하면 됩니다; 딕셔너리와 시퀀스가 잘 작동합니다. 매핑되지 않은 문자 서수(`LookupError`를 유발하는 것)는 건드리지 않고 그대로 복사됩니다.

*errors*는 코덱에서의 일반적인 의미입니다. 기본 에러 처리를 사용함을 나타내는 `NULL`일 수 있습니다.

윈도우 용 MBCS 코덱

다음은 MBCS 코덱 API입니다. 현재 윈도우에서만 사용할 수 있으며 Win32 MBCS 변환기를 사용하여 변환을 구현합니다. MBCS(또는 DBCS)는 단지 하나가 아니라 인코딩 클래스임에 유의하십시오. 대상 인코딩은 코덱을 실행하는 기계의 사용자 설정에 의해 정의됩니다.

PyObject ***PyUnicode_DecodeMBCS**(const char *str, *Py_ssize_t* size, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. MBCS 인코딩된 문자열 *str*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 `NULL`을 반환합니다.

PyObject ***PyUnicode_DecodeMBCSStateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

반환값: 새 참조. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. *consumed*가 `NULL`이면, *PyUnicode_DecodeMBCS()*처럼 동작합니다. *consumed*가 `NULL`이 아니면, *PyUnicode_DecodeMBCSStateful()*은 후행 선행(lead) 바이트를 디코딩하지 않고 디코딩된 바이트 수가 *consumed*에 저장됩니다.

PyObject ***PyUnicode_DecodeCodePageStateful**(int code_page, const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

반환값: 새 참조. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. Similar to *PyUnicode_DecodeMBCSStateful()*, except uses the code page specified by *code_page*.

PyObject ***PyUnicode_AsMBCSString**(*PyObject* *unicode)

반환값: 새 참조. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. MBCS를 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 "strict" 입니다. 코덱에서 예외가 발생하면 `NULL`을 반환합니다.

PyObject ***PyUnicode_EncodeCodePage**(int code_page, *PyObject* *unicode, const char *errors)

반환값: 새 참조. *Part of the* 안정 ABI *on Windows* 버전 *3.7* 이후로. 지정된 코드 페이지를 사용하여 유니코드 객체를 인코딩하고 파이썬 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 `NULL`을 반환합니다. `CP_ACP` 코드 페이지를 사용하여 MBCS 인코더를 얻습니다.

Added in version 3.3.

메서드와 슬롯 함수

다음 API는 입력의 유니코드 객체와 문자열을 (설명에서 문자열이라고 하겠습니다) 처리할 수 있으며 적절하게 유니코드 객체나 정수를 반환합니다.

예외가 발생하면 모두 `NULL`이나 `-1`을 반환합니다.

PyObject ***PyUnicode_Concat**(*PyObject* *left, *PyObject* *right)

반환값: 새 참조. *Part of the* 안정 ABI. 두 문자열을 이어붙여 하나의 새로운 유니코드 문자열을 제공합니다.

PyObject ***PyUnicode_Split**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

반환값: 새 참조. *Part of the* 안정 ABI. 문자열을 분할하여 유니코드 문자열 리스트를 제공합니다. *sep* 이 `NULL`이면, 모든 공백 부분 문자열에서 분할이 수행됩니다. 그렇지 않으면, 주어진 구분자에서 분할이 일어납니다. 최대 *maxsplit* 분할이 수행됩니다. 음수이면, 제한이 설정되지 않습니다. 구분자는 결과 리스트에 포함되지 않습니다.

에러 시, 예외를 설정하고 `NULL`을 반환합니다.

Equivalent to `str.split()`.

PyObject ***PyUnicode_RSplit**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

반환값: 새 참조. *Part of the* 안정 ABI. Similar to *PyUnicode_Split()*, but splitting will be done beginning at the end of the string.

에러 시, 예외를 설정하고 `NULL`을 반환합니다.

Equivalent to `str.rsplit()`.

PyObject ***PyUnicode_Splitlines**(*PyObject* *unicode, int keepends)

반환값: 새 참조. *Part of the* 안정 ABI. 줄 바꿈에서 유니코드 문자열을 분할하여, 유니코드 문자열 리스트를 반환합니다. CRLF는 하나의 줄 바꿈으로 간주합니다. *keepends*가 `0`이면, 결과 문자열에 줄 바꿈 문자가 포함되지 않습니다.

PyObject ***PyUnicode_Partition**(*PyObject* *unicode, *PyObject* *sep)

반환값: 새 참조. *Part of the* 안정 ABI. Split a Unicode string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

sep must not be empty.

에러 시, 예외를 설정하고 `NULL`을 반환합니다.

Equivalent to `str.partition()`.

PyObject ***PyUnicode_RPartition**(*PyObject* *unicode, *PyObject* *sep)

반환값: 새 참조. *Part of the* 안정 ABI. Similar to *PyUnicode_Partition()*, but split a Unicode string at the last occurrence of *sep*. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

sep must not be empty.

에러 시, 예외를 설정하고 `NULL`을 반환합니다.

Equivalent to `str.rpartition()`.

PyObject ***PyUnicode_Join**(*PyObject* *separator, *PyObject* *seq)

반환값: 새 참조. *Part of the* 안정 ABI. 주어진 *separator*를 사용하여 문자열 시퀀스를 연결하고 결과 유니코드 문자열을 반환합니다.

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the 안정 ABI. *substr*이 주어진 꼬리 끝에서 (*direction* == `-1`은 접두사 일치를 수행함을 의미하고, *direction* == `1`은 접미사 일치를 의미합니다) `unicode[start:end]`와 일치하면 `1`을 반환합니다, 그렇지 않으면 `0`을 반환합니다. 에러가 발생하면 `-1`을 반환합니다.

Py_ssize_t **PyUnicode_Find**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the 안정 ABI. 주어진 *direction*을 사용하여 (*direction* == `1`은 정방향 검색을 의미하고, *direction* == `-1`은 역방향 검색을 의미합니다) `unicode[start:end]`에서 *substr*의 첫 번째 위치를 반환합니다. 반환 값은 첫 번째 일치의 인덱스입니다; `-1` 값은 일치하는 항목이 없음을 나타내고, `-2`는 에러가 발생했고 예외가 설정되었음을 나타냅니다.

Py_ssize_t **PyUnicode_FindChar**(*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the 안정 ABI 버전 *3.7* 이후로. 주어진 *direction*을 사용하여 (*direction* == `1`은 정방향 검색을 의미하고, *direction* == `-1`은 역방향 검색을 의미합니다) `unicode[start:end]`에서 문자 *ch*의 첫 번

째 위치를 반환합니다. 반환 값은 첫 번째 일치의 인덱스입니다; `-1` 값은 일치하는 항목이 없음을 나타내고, `-2`는 에러가 발생했고 예외가 설정되었음을 나타냅니다.

Added in version 3.3.

버전 3.7에서 변경: *start*와 *end*는 이제 `unicode[start:end]`처럼 작동하도록 조정됩니다.

Py_ssize_t **PyUnicode_Count**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Part of the 안정 ABI. `unicode[start:end]`에서 *substr*이 겹치지 않게 등장하는 횟수를 반환합니다. 에러가 발생하면 `-1`을 반환합니다.

PyObject ***PyUnicode_Replace**(*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

반환값: 새 참조. *Part of the* 안정 ABI. *unicode*에서 *substr*의 최대 *maxcount* 등장을 *replstr*로 바꾸고 결과 유니코드 객체를 반환합니다. *maxcount* == `-1`은 모든 등장을 교체함을 의미합니다.

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)

Part of the 안정 ABI. 두 문자열을 비교하고 각각 작음, 같음, 큼에 대해 `-1`, `0`, `1`을 반환합니다.

이 함수는 실패 시 `-1`을 반환하므로, 에러를 확인하기 위해 *PyErr_Occurred()*를 호출해야 합니다.

더 보기

The *PyUnicode_Equal()* function.

int **PyUnicode_Equal**(*PyObject* *a, *PyObject* *b)

Part of the 안정 ABI 버전 *3.14* 이후로. Test if two strings are equal:

- Return `1` if *a* is equal to *b*.
- Return `0` if *a* is not equal to *b*.
- Set a `TypeError` exception and return `-1` if *a* or *b* is not a `str` object.

The function always succeeds if *a* and *b* are `str` objects.

The function works for `str` subclasses, but does not honor custom `__eq__()` method.

더 보기

The *PyUnicode_Compare()* function.

Added in version 3.14.

int **PyUnicode_EqualToUTF8AndSize**(*PyObject* *unicode, const char *string, *Py_ssize_t* size)

Part of the 안정 ABI 버전 *3.13* 이후로. Compare a Unicode object with a char buffer which is interpreted as being UTF-8 or ASCII encoded and return true (`1`) if they are equal, or false (`0`) otherwise. If the Unicode object contains surrogate code points (`U+D800` - `U+DFFF`) or the C string is not valid UTF-8, false (`0`) is returned.

이 함수는 예외를 발생시키지 않습니다.

Added in version 3.13.

int **PyUnicode_EqualToUTF8**(*PyObject* *unicode, const char *string)

Part of the 안정 ABI 버전 *3.13* 이후로. *PyUnicode_EqualToUTF8AndSize()*와 유사하지만, `strlen()`을 사용하여 *string* 길이를 계산합니다. 유니코드 객체가 널 문자를 포함하면, 거짓(`0`)을 반환합니다.

Added in version 3.13.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

Part of the 안정 ABI. 유니코드 객체 *unicode*를 *string*과 비교하고 각각 작음, 같음, 큼에 대해 `-1`, `0`, `1`을 반환합니다. ASCII로 인코딩된 문자열만 전달하는 것이 가장 좋지만, 비 ASCII 문자가 포함되면 함수는 입력 문자열을 ISO-8859-1로 해석합니다.

이 함수는 예외를 발생시키지 않습니다.

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

반환값: 새 참조. *Part of the* 안정 ABI. 두 유니코드 문자열을 풍부한 비교(rich comparison) 하고 다음 중 하나를 반환합니다:

- 예외가 발생하면 `NULL`
- 성공적인 비교는 *Py_True*나 *Py_False*
- 형 조합을 알 수 없으면 *Py_NotImplemented*

*op*에 가능한 값은 *Py_GT*, *Py_GE*, *Py_EQ*, *Py_NE*, *Py_LT* 및 *Py_LE*입니다.

PyObject ***PyUnicode_Format**(*PyObject* *format, *PyObject* *args)

반환값: 새 참조. *Part of the* 안정 ABI. *format*과 *args*에서 새 문자열 객체를 반환합니다; 이것은 `format % args`와 유사합니다.

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

Part of the 안정 ABI. *substr*가 *unicode*에 포함되어 있는지 확인하고 그에 따라 참이나 거짓을 반환합니다.

*substr*는 단일 요소 유니코드 문자열로 강제 변환해야 합니다. 에러가 있으면 `-1`이 반환됩니다.

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

Part of the 안정 ABI. 인자 **p_unicode*를 제자리에서 인턴(intern) 합니다. 인자는 파이썬 유니코드 문자열을 가리키는 포인터 변수의 주소여야 합니다. **p_unicode*와 같은 기존 인턴 문자열이 있으면, **p_unicode*를 그것으로 설정합니다 (이전 문자열 객체에 대한 참조를 해제하고 인턴 된 문자열 객체에 대한 새로운 강한 참조를 만듭니다), 그렇지 않으면 **p_unicode*만 홀로 두고 인턴 합니다.

(설명: 참조에 대해 많은 이야기가 있지만, 이 함수를 참조 중립이라고 생각하십시오. 전달하는 객체의 소유권을 반드시 가져야 합니다; 호출 후 더는 전달된 참조를 소유하지 않지만, 결과를 새로 소유하게 됩니다.)

This function never raises an exception. On error, it leaves its argument unchanged without interning it.

Instances of subclasses of `str` may not be interned, that is, *PyUnicode_CheckExact*(**p_unicode*) must be true. If it is not, then – as with any other error – the argument is left unchanged.

Note that interned strings are not "immortal". You must keep a reference to the result to benefit from interning.

PyObject ***PyUnicode_InternFromString**(const char *str)

반환값: 새 참조. *Part of the* 안정 ABI. *PyUnicode_FromString()*과 *PyUnicode_InternInPlace()*의 조합, 정적 할당된 문자열을 위한 것입니다.

인턴(intern) 된 새 유니코드 문자열 객체나, 같은 값을 가진 이전에 인턴 된 문자열 객체에 대한 새 ("소유된") 참조를 반환합니다.

Python may keep a reference to the result, or make it *immortal*, preventing it from being garbage-collected promptly. For interning an unbounded number of different strings, such as ones coming from user input, prefer calling *PyUnicode_FromString()* and *PyUnicode_InternInPlace()* directly.

unsigned int **PyUnicode_CHECK_INTERNED**(*PyObject* *str)

Return a non-zero value if *str* is interned, zero if not. The *str* argument must be a string; this is not checked. This function always succeeds.

CPython 구현 상세: A non-zero return value may carry additional information about *how* the string is interned. The meaning of such non-zero values, as well as each specific string's intern-related details, may change between CPython versions.

PyUnicodeWriter

The *PyUnicodeWriter* API can be used to create a Python `str` object.

Added in version 3.14.

type **PyUnicodeWriter**

A Unicode writer instance.

The instance must be destroyed by *PyUnicodeWriter_Finish()* on success, or *PyUnicodeWriter_Discard()* on error.

PyUnicodeWriter ***PyUnicodeWriter_Create**(*Py_ssize_t* length)

Create a Unicode writer instance.

length must be greater than or equal to `0`.

If *length* is greater than `0`, preallocate an internal buffer of *length* characters.

Set an exception and return `NULL` on error.

PyObject ***PyUnicodeWriter_Finish**(*PyUnicodeWriter* *writer)

Return the final Python `str` object and destroy the writer instance.

Set an exception and return `NULL` on error.

The writer instance is invalid after this call.

void **PyUnicodeWriter_Discard**(*PyUnicodeWriter* *writer)

Discard the internal Unicode buffer and destroy the writer instance.

If *writer* is `NULL`, no operation is performed.

The writer instance is invalid after this call.

int **PyUnicodeWriter_WriteChar**(*PyUnicodeWriter* *writer, *Py_UCS4* ch)

Write the single Unicode character *ch* into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteUTF8**(*PyUnicodeWriter* *writer, const char *str, *Py_ssize_t* size)

Decode the string *str* from UTF-8 in strict mode and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also *PyUnicodeWriter_DecodeUTF8Stateful()*.

int **PyUnicodeWriter_WriteASCII**(*PyUnicodeWriter* *writer, const char *str, *Py_ssize_t* size)

Write the ASCII string *str* into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

str must only contain ASCII characters. The behavior is undefined if *str* contains non-ASCII characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

Added in version 3.14.

int **PyUnicodeWriter_WriteWideChar**(*PyUnicodeWriter* *writer, const wchar_t *str, *Py_ssize_t* size)

Write the wide string *str* into *writer*.

size is a number of wide characters. If *size* is equal to `-1`, call `wcslen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteUCS4**(*PyUnicodeWriter* *writer, *Py_UCS4* *str, *Py_ssize_t* size)

Writer the UCS4 string *str* into *writer*.

size is a number of UCS4 characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteStr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call `PyObject_Str()` on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteRepr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call `PyObject_Repr()` on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteSubstring**(*PyUnicodeWriter* *writer, *PyObject* *str, *Py_ssize_t* start, *Py_ssize_t* end)

Write the substring `str[start:end]` into *writer*.

str must be Python `str` object. *start* must be greater than or equal to 0, and less than or equal to *end*. *end* must be less than or equal to *str* length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_Format**(*PyUnicodeWriter* *writer, const char *format, ...)

Similar to `PyUnicode_FromFormat()`, but write the output directly into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_DecodeUTF8Stateful**(*PyUnicodeWriter* *writer, const char *string, *Py_ssize_t* length, const char *errors, *Py_ssize_t* *consumed)

Decode the string *str* from UTF-8 with *errors* error handler and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

errors is an error handler name, such as `"replace"`. If *errors* is `NULL`, use the strict error handler.

If *consumed* is not `NULL`, set **consumed* to the number of decoded bytes on success. If *consumed* is `NULL`, treat trailing incomplete UTF-8 byte sequences as an error.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also `PyUnicodeWriter_WriteUTF8()`.

Deprecated API

The following API is deprecated.

type **Py_UNICODE**

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform. Please use `wchar_t` directly instead.

버전 3.3에서 변경: 이전 버전에서, 이것은 빌드 시 파이썬의 "내로우(narrow)" 나 "와이드(wide)" 유니코드 버전 중 어느 것을 선택했는지에 따라 16비트 형이나 32비트 형이었습니다.

Deprecated since version 3.13, will be removed in version 3.15.

int **PyUnicode_READY**(*PyObject* *unicode)

Do nothing and return `0`. This API is kept only for backward compatibility, but there are no plans to remove it.

Added in version 3.3.

버전 3.10부터 폐지됨: This API does nothing since Python 3.12. Previously, this needed to be called for each string created using the old API (`PyUnicode_FromUnicode()` or similar).

unsigned int **PyUnicode_IS_READY**(*PyObject* *unicode)

Do nothing and return `1`. This API is kept only for backward compatibility, but there are no plans to remove it.

Added in version 3.3.

버전 3.14부터 폐지됨: This API does nothing since Python 3.12. Previously, this could be called to check if *PyUnicode_READY()* is necessary.

9.3.4 튜플 객체

type **PyTupleObject**

이 *PyObject*의 서브 형은 파이썬 튜플 객체를 나타냅니다.

PyTypeObject **PyTuple_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 튜플 형을 나타냅니다. 파이썬 계층의 `tuple`과 같은 객체입니다.

int **PyTuple_Check**(*PyObject* *p)

*p*가 튜플 객체이거나 튜플 형의 서브 형의 인스턴스면 참을 돌려줍니다. 이 함수는 항상 성공합니다.

int **PyTuple_CheckExact**(*PyObject* *p)

*p*가 튜플 객체이지만, 튜플 형의 서브 형의 인스턴스는 아니면 참을 돌려줍니다. 이 함수는 항상 성공합니다.

PyObject ***PyTuple_New**(*Py_ssize_t* len)

반환값: 새 참조. *Part of the* 안정 ABI. 크기 *len* 인 튜플 객체나, 실패 시 예외를 설정하고 `NULL`을 반환합니다.

PyObject ***PyTuple_Pack**(*Py_ssize_t* n, ...)

반환값: 새 참조. *Part of the* 안정 ABI. 크기 *n* 인 새 튜플 객체나, 실패 시 예외를 설정하고 `NULL`을 반환합니다. 튜플 값은 파이썬 객체를 가리키는 후속 *n* 개의 C 인자로 초기화됩니다. `PyTuple_Pack(2, a, b)`는 `Py_BuildValue("(OO)", a, b)`와 동등합니다.

Py_ssize_t **PyTuple_Size**(*PyObject* *p)

Part of the 안정 ABI. 튜플 객체에 대한 포인터를 받아서, 해당 튜플의 크기를 반환합니다. 에러가 발생하면, 예외를 설정하고 `-1`을 반환합니다.

Py_ssize_t **PyTuple_GET_SIZE**(*PyObject* *p)

*PyTuple_Size()*와 비슷하지만, 에러 검사를 생략합니다.

PyObject ***PyTuple_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

반환값: 빌린 참조. *Part of the* 안정 ABI. *p*가 가리키는 튜플의 *pos* 위치에 있는 객체를 반환합니다. *pos*가 음수이거나 범위를 벗어나면, `NULL`을 반환하고 `IndexError` 예외를 설정합니다.

The returned reference is borrowed from the tuple *p* (that is: it is only valid as long as you hold a reference to *p*). To get a *strong reference*, use *Py_NewRef(PyTuple_GetItem(...))* or *PySequence_GetItem()*.

PyObject ***PyTuple_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

반환값: 빌린 참조. *PyTuple_GetItem()*와 비슷하지만, 인자를 확인하지 않습니다.

PyObject ***PyTuple_GetSlice**(*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

반환값: 새 참조. *Part of the* 안정 ABI. *p*가 가리키는 튜플의 *low*와 *high* 사이의 슬라이스를 반환하거나, 실패하면 예외를 설정하고 `NULL`을 반환합니다.

이것은 파이썬 표현식 `p[low:high]`와 동등합니다. 튜플 끝으로부터의 인덱싱은 지원되지 않습니다.

int **PyTuple_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Part of the 안정 ABI. *p*가 가리키는 튜플의 *pos* 위치에 객체 *o*에 대한 참조를 삽입합니다. 성공하면 `0`을 반환합니다. *pos*가 범위를 벗어나면, `-1`을 반환하고 `IndexError` 예외를 설정합니다.

> **참고**
>
> 이 함수는 *o*에 대한 참조를 "훔치고" 영향을 받는 위치에서 튜플에 이미 있는 항목에 대한 참조를 버립니다(discard).

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

*PyTuple_SetItem()*과 비슷하지만, 에러 검사는 하지 않으며 새로운 튜플을 채울 때*만* 사용해야 합니다.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **참고**
>
> 이 함수는 *o*에 대한 참조를 "훔치고", *PyTuple_SetItem()*와 달리, 교체 중인 항목에 대한 참조를 버리지 않습니다; *pos* 위치에서 튜플의 모든 참조는 누수됩니다.

> **경고**
>
> This macro should *only* be used on tuples that are newly created. Using this macro on a tuple that is already in use (or in other words, has a refcount > 1) could lead to undefined behavior.

int **_PyTuple_Resize**(*PyObject* **p, *Py_ssize_t* newsize)

튜플 크기를 조정하는 데 사용할 수 있습니다. *newsize*는 튜플의 새로운 길이가 됩니다. 튜플은 불변이라고 여겨지므로, 객체에 대해 참조가 하나만 있을 때만 사용해야 합니다. 튜플이 코드의 다른 부분에 이미 알려졌으면 이것을 사용하지 마십시오. 튜플은 항상 끝에서 커지거나 줄어듭니다. 이것을 오래된 튜플을 파괴하고 새 튜플을 만드는 것으로 생각하십시오, 단지 더 효율적일 뿐입니다. 성공하면 `0`을 반환합니다. 클라이언트 코드는, `*p`의 결괏값이 이 함수를 호출하기 전과 같다고 가정해서는 안 됩니다. `*p`가 참조하는 객체가 바뀌면 원래 `*p`는 파괴됩니다. 실패하면, `-1`을 반환하고, `*p`를 `NULL`로 설정하고, `MemoryError` 나 `SystemError`를 발생시킵니다.

9.3.5 구조체 시퀀스 객체

구조체 시퀀스(struct sequence) 객체는 `namedtuple()` 객체의 C 등가물입니다, 즉 어트리뷰트를 통해 항목에 액세스할 수 있는 시퀀스입니다. 구조체 시퀀스를 만들려면, 먼저 특정 구조체 시퀀스 형을 만들어야 합니다.

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

반환값: 새 참조. *Part of the* 안정 ABI. 아래에 설명된 *desc*의 데이터로 새로운 구조체 시퀀스 형을 만듭니다. 결과 형의 인스턴스는 *PyStructSequence_New()*로 만들 수 있습니다.

실패 시 예외를 설정하고 `NULL`을 반환합니다.

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

*desc*로 구조체 시퀀스 형 *type*을 재자리에서 초기화합니다.

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

*PyStructSequence_InitType()*와 비슷하지만, 성공하면 `0`을, 실패하면 예외를 설정하고 `-1`을 반환합니다.

Added in version 3.4.

type **PyStructSequence_Desc**

Part of the 안정 ABI (*including all members*). 만들 구조체 시퀀스 형의 메타 정보를 포함합니다.

const char ***name**

Fully qualified name of the type; null-terminated UTF-8 encoded. The name must contain the module name.

const char ***doc**

해당 형에 대한 독스트링에 대한 포인터나 생략하려면 `NULL`.

PyStructSequence_Field ***fields**

새로운 형의 필드 이름을 가진 `NULL`로 끝나는 배열에 대한 포인터.

int **n_in_sequence**

파이썬 측에서 볼 수 있는 필드 수 (튜플로 사용된 경우).

type **PyStructSequence_Field**

Part of the 안정 ABI (*including all members*). 구조체 시퀀스의 필드를 기술합니다. 구조체 시퀀스는 튜플로 모형화되므로, 모든 필드는 *PyObject** 형을 취합니다. *PyStructSequence_Desc*의 *fields* 배열의 인덱스는 구조체 시퀀스의 어떤 필드가 기술되는지를 결정합니다.

const char ***name**

필드의 이름. 또는 이름있는 필드의 목록을 끝내려면 NULL, 이름이 없는 상태로 두려면 *PyStructSequence_UnnamedField*로 설정합니다.

const char ***doc**

필드 독스트링이나 생략하려면 NULL.

const char *const **PyStructSequence_UnnamedField**

Part of the 안정 ABI 버전 *3.11* 이후로. 이름 없는 상태로 남겨두기 위한 필드 이름의 특수 값.

버전 3.9에서 변경: 형이 `char *`에서 변경되었습니다.

PyObject ***PyStructSequence_New**(*PyTypeObject* *type)

반환값: 새 참조. *Part of the* 안정 ABI. *PyStructSequence_NewType()*으로 만든 *type*의 인스턴스를 만듭니다.

실패 시 예외를 설정하고 NULL을 반환합니다.

PyObject ***PyStructSequence_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

반환값: 빌린 참조. *Part of the* 안정 ABI. *p*가 가리키는 구조체 시퀀스의 위치 *pos*에 있는 객체를 돌려줍니다.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

PyObject ***PyStructSequence_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

반환값: 빌린 참조. *PyStructSequence_GetItem()*의 별칭.

버전 3.13에서 변경: 이제 *PyStructSequence_GetItem()*의 별칭으로 구현됩니다.

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Part of the 안정 ABI. 구조체 시퀀스 *p*의 인덱스 *pos*에 있는 필드를 값 *o*로 설정합니다. *PyTuple_SET_ITEM()*과 마찬가지로, 이것은 새로운 인스턴스를 채울 때만 사용해야 합니다.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **참고**
>
> 이 함수는 *o*에 대한 참조를 "훔칩니다".

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

*PyStructSequence_SetItem()*의 별칭.

버전 3.13에서 변경: 이제 *PyStructSequence_SetItem()*의 별칭으로 구현됩니다.

9.3.6 리스트 객체

type **PyListObject**

이 *PyObject*의 서브 형은 파이썬 리스트 객체를 나타냅니다.

PyTypeObject **PyList_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 리스트 형을 나타냅니다. 이것은 파이썬 계층의 `list`와 같은 객체입니다.

int **PyList_Check**(*PyObject* *p)

*p*가 리스트 객체나 리스트 형의 서브 형 인스턴스면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyList_CheckExact**(*PyObject* *p)

*p*가 리스트 객체이지만 리스트 형의 서브 형의 인스턴스가 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyList_New**(*Py_ssize_t* len)

반환값: 새 참조. *Part of the* 안정 ABI. 성공하면 길이 *len* 인 새 리스트를, 실패하면 `NULL`을 반환합니다.

> **참고**
>
> *len*이 0보다 크면, 반환된 리스트 객체의 항목은 `NULL`로 설정됩니다. 따라서 모든 항목을 `PyList_SetItem()`이나 `PyList_SET_ITEM()`으로 실제 객체로 설정하기 전에 `PySequence_SetItem()`와 같은 추상 API 함수를 사용하거나 파이썬 코드에 객체를 노출할 수 없습니다. 다음 API는 리스트가 완전히 초기화되기 전의 안전한 API 입니다: `PyList_SetItem()`과 `PyList_SET_ITEM()`.

Py_ssize_t **PyList_Size**(*PyObject* *list)

Part of the 안정 ABI. *list*에서 리스트 객체의 길이를 반환합니다; 이는 리스트 객체에 대한 `len(list)`와 동등합니다.

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

`PyList_Size()`와 유사하지만, 에러 검사가 없습니다.

PyObject ***PyList_GetItemRef**(*PyObject* *list, *Py_ssize_t* index)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.13* 이후로. *list*가 가리키는 리스트에서 *index* 위치의 객체를 반환합니다. 위치는 음수가 아니어야 합니다; 리스트의 끝에서부터의 인덱싱은 지원되지 않습니다. *index*가 범위를 벗어나면 (`<0 or >=len(list)`), `NULL`을 반환하고 `IndexError` 예외를 설정합니다.

Added in version 3.13.

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

반환값: 빌린 참조. *Part of the* 안정 ABI. Like `PyList_GetItemRef()`, but returns a *borrowed reference* instead of a *strong reference*.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

반환값: 빌린 참조. `PyList_GetItem()`와 유사하지만, 에러 검사가 없습니다.

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Part of the 안정 ABI. 리스트의 인덱스 *index*에 있는 항목을 *item*으로 설정합니다. 성공하면 `0`을 반환합니다. *index*가 범위를 벗어나면, `-1`을 반환하고 `IndexError` 예외를 설정합니다.

> **참고**
>
> 이 함수는 *item*에 대한 참조를 "훔치고" 영향을 받는 위치의 리스트에 이미 있는 항목에 대한 참조를 버립니다.

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

에러 검사 없는 `PyList_SetItem()`의 매크로 형식. 일반적으로 이전 내용이 없는 새 리스트를 채우는 데 사용됩니다.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **참고**
>
> 이 매크로는 *item*에 대한 참조를 "훔치고", `PyList_SetItem()`과는 달리 대체되는 항목에 대한 참조를 버리지 않습니다; *list*의 *i* 위치에 있는 참조는 누수를 일으킵니다.

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Part of the 안정 ABI. 항목 *item*을 리스트 *list*의 인덱스 *index* 앞에 삽입합니다. 성공하면 `0`을 반환합니다; 실패하면 `-1`을 반환하고 예외를 설정합니다. `list.insert(index, item)`에 해당합니다.

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

Part of the 안정 ABI. 리스트 *list*의 끝에 객체 *item*을 추가합니다. 성공하면 `0`을 반환합니다; 실패하면 `-1`을 반환하고 예외를 설정합니다. `list.append(item)`에 해당합니다.

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

반환값: 새 참조. *Part of the* 안정 ABI. *list*에서 *low*와 *high* 사이에있는 객체들을 포함하는 리스트를 반환합니다. 실패하면 `NULL`을 반환하고 예외를 설정합니다. `list[low:high]`에 해당합니다. 리스트 끝에서부터의 인덱싱은 지원되지 않습니다.

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

Part of the 안정 ABI. *low*와 *high* 사이의 *list* 슬라이스를 *itemlist*의 내용으로 설정합니다. `list[low:high] = itemlist`에 해당합니다. *itemlist*는 `NULL`일 수 있는데, 빈 리스트의 대입을 나타냅니다 (슬라이스 삭제). 성공하면 `0`을, 실패하면 `-1`을 반환합니다. 리스트 끝에서부터의 인덱싱은 지원되지 않습니다.

int **PyList_Extend**(*PyObject* *list, *PyObject* *iterable)

Extend *list* with the contents of *iterable*. This is the same as `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` and analogous to `list.extend(iterable)` or `list += iterable`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

Added in version 3.13.

int **PyList_Clear**(*PyObject* *list)

Remove all items from *list*. This is the same as `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` and analogous to `list.clear()` or `del list[:]`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

Added in version 3.13.

int **PyList_Sort**(*PyObject* *list)

Part of the 안정 ABI. *list* 항목을 제자리에서 정렬합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다. 이것은 `list.sort()`와 동등합니다.

int **PyList_Reverse**(*PyObject* *list)

Part of the 안정 ABI. *list*의 항목을 제자리에서 뒤집습니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다. 이것은 `list.reverse()`와 동등합니다.

PyObject ***PyList_AsTuple**(*PyObject* *list)

반환값: 새 참조. *Part of the* 안정 ABI. *list*의 내용을 포함하는 새 튜플 객체를 반환합니다; `tuple(list)`와 동등합니다.

9.4 컨테이너 객체

9.4.1 딕셔너리 객체

type **PyDictObject**

이 `PyObject`의 서브 형은 파이썬 딕셔너리 객체를 나타냅니다.

PyTypeObject **PyDict_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 딕셔너리 형을 나타냅니다. 이것은 파이썬 계층의 `dict`와 같은 객체입니다.

int **PyDict_Check**(*PyObject* *p)

*p*가 dict 객체이거나 dict 형의 서브 형의 인스턴스면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyDict_CheckExact**(*PyObject* *p)

*p*가 dict 객체이지만, dict 형의 서브 형의 인스턴스는 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyDict_New**()

반환값: 새 참조. *Part of the* 안정 ABI. 새로운 빈 딕셔너리를 반환하거나, 실패하면 `NULL`을 반환합니다.

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

반환값: 새 참조. *Part of the* 안정 ABI. 읽기 전용 동작을 강제하는 매핑을 위한 `types.MappingProxyType` 객체를 반환합니다. 이것은 일반적으로 비 동적 클래스 형을 위한 딕셔너리의 수정을 방지하기 위해 뷰를 만드는 데 사용됩니다.

void **PyDict_Clear**(*PyObject* *p)

Part of the 안정 ABI. 기존 딕셔너리의 모든 키-값 쌍을 비웁니다.

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

Part of the 안정 ABI. 딕셔너리 *p*에 *key*가 포함되어 있는지 확인합니다. *p*의 항목이 *key*와 일치하면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. 에러면 `-1`을 반환합니다. 이는 파이썬 표현식 `key in p`와 동등합니다.

int **PyDict_ContainsString**(*PyObject* *p, const char *key)

이것은 *PyDict_Contains()*와 같지만, *key*가 *PyObject** 가 아닌 `const char*` UTF-8 인코딩된 바이트 문자열로 지정됩니다.

Added in version 3.13.

PyObject ***PyDict_Copy**(*PyObject* *p)

반환값: 새 참조. *Part of the* 안정 ABI. *p*와 같은 키-값 쌍을 포함하는 새 딕셔너리를 반환합니다.

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

Part of the 안정 ABI. 딕셔너리 *p*에 *val*을 *key* 키로 삽입합니다. *key*는 해시 가능해야 합니다. 그렇지 않으면 `TypeError`가 발생합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다. 이 함수는 *val*에 대한 참조를 훔치지 않습니다.

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)

Part of the 안정 ABI. 이것은 *PyDict_SetItem()*와 같지만, *key*가 *PyObject** 가 아닌 `const char*` UTF-8 인코딩된 바이트 문자열로 지정됩니다.

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)

Part of the 안정 ABI. 딕셔너리 *p*에서 키가 *key*인 항목을 제거합니다. *key*는 해시 가능해야 합니다. 그렇지 않으면 `TypeError`가 발생합니다. *key*가 딕셔너리에 없으면, `KeyError`가 발생합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

Part of the 안정 ABI. 이것은 *PyDict_DelItem()*와 같지만, *key*가 *PyObject** 가 아닌 `const char*` UTF-8 인코딩된 바이트 문자열로 지정됩니다.

int **PyDict_GetItemRef**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Part of the 안정 ABI 버전 *3.13* 이후로. Return a new *strong reference* to the object from dictionary *p* which has a key *key*:

- If the key is present, set **result* to a new *strong reference* to the value and return `1`.
- If the key is missing, set **result* to `NULL` and return `0`.

- On error, raise an exception and return `-1`.

Added in version 3.13.

See also the *PyObject_GetItem()* function.

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

반환값: 빌린 참조. *Part of the* 안정 ABI. 딕셔너리 *p*에서 키가 *key*인 객체에 대한 빌린 참조를 반환합니다. *key* 키가 없으면 예외를 설정하지 않고 `NULL`을 반환합니다.

> **참고**
>
> `__hash__()`와 `__eq__()` 메서드를 호출하는 동안 발생하는 예외는 조용히 무시됩니다. 대신 *PyDict_GetItemWithError()* 함수를 사용하십시오.

버전 3.10에서 변경: Calling this API without an *attached thread state* had been allowed for historical reason. It is no longer allowed.

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

반환값: 빌린 참조. *Part of the* 안정 ABI. 예외를 억제하지 않는 *PyDict_GetItem()*의 변형입니다. 예외가 발생하면 예외를 **설정하고** `NULL`을 반환합니다. 키가 없으면 예외를 설정하지 **않고** `NULL`을 반환합니다.

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

반환값: 빌린 참조. *Part of the* 안정 ABI. 이것은 *PyDict_GetItem()*와 같지만, *key*가 *PyObject** 가 아닌 `const char*` UTF-8로 인코딩된 바이트 문자열로 지정됩니다.

> **참고**
>
> `__hash__()`와 `__eq__()` 메서드를 호출하는 동안이나 임시 `str` 객체를 만드는 동안 발생하는 예외는 조용히 무시됩니다. 대신 자체 *PyUnicode_FromString()* *key*와 함께 *PyDict_GetItemWithError()* 함수를 사용하는 것이 좋습니다.

int **PyDict_GetItemStringRef**(*PyObject* *p, const char *key, *PyObject* **result)

Part of the 안정 ABI 버전 *3.13* 이후로. 이것은 *PyDict_GetItemRef()*와 유사하지만, *key*가 *PyObject** 가 아닌 `const char*` UTF-8로 인코딩된 바이트 문자열로 지정됩니다.

Added in version 3.13.

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

반환값: 빌린 참조. 이것은 파이썬 수준의 `dict.setdefault()`와 같습니다. 존재하면, 딕셔너리 *p*에서 *key*에 해당하는 값을 반환합니다. 키가 dict에 없으면, 값 *defaultobj*로 삽입되고, *defaultobj*가 반환됩니다. 이 함수는 *key*의 해시 함수를 조회 및 삽입을 위해 독립적으로 평가하는 대신 한 번만 평가합니다.

Added in version 3.4.

int **PyDict_SetDefaultRef**(*PyObject* *p, *PyObject* *key, *PyObject* *default_value, *PyObject* **result)

Inserts *default_value* into the dictionary *p* with a key of *key* if the key is not already present in the dictionary. If *result* is not `NULL`, then **result* is set to a *strong reference* to either *default_value*, if the key was not present, or the existing value, if *key* was already present in the dictionary. Returns `1` if the key was present and *default_value* was not inserted, or `0` if the key was not present and *default_value* was inserted. On failure, returns `-1`, sets an exception, and sets `*result` to `NULL`.

For clarity: if you have a strong reference to *default_value* before calling this function, then after it returns, you hold a strong reference to both *default_value* and **result* (if it' s not `NULL`). These may refer to the same object: in that case you hold two separate references to it.

Added in version 3.13.

int **PyDict_Pop**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Remove *key* from dictionary *p* and optionally return the removed value. Do not raise `KeyError` if the key missing.

- If the key is present, set **result* to a new reference to the removed value if *result* is not `NULL`, and return `1`.
- If the key is missing, set **result* to `NULL` if *result* is not `NULL`, and return `0`.
- On error, raise an exception and return `-1`.

Similar to `dict.pop()`, but without the default value and not raising `KeyError` if the key missing.

Added in version 3.13.

int **PyDict_PopString**(*PyObject* *p, const char *key, *PyObject* **result)

이것은 *PyDict_Pop()*와 유사하지만, *key*가 *PyObject** 가 아닌 `const char*` UTF-8로 인코딩된 바이트 문자열로 지정됩니다.

Added in version 3.13.

PyObject ***PyDict_Items**(*PyObject* *p)

반환값: 새 참조. *Part of the* 안정 ABI. 딕셔너리의 모든 항목을 포함하는 *PyListObject*를 반환합니다.

PyObject ***PyDict_Keys**(*PyObject* *p)

반환값: 새 참조. *Part of the* 안정 ABI. 딕셔너리의 모든 키를 포함하는 *PyListObject*를 반환합니다.

PyObject ***PyDict_Values**(*PyObject* *p)

반환값: 새 참조. *Part of the* 안정 ABI. 딕셔너리 *p*의 모든 값을 포함하는 *PyListObject*를 반환합니다.

Py_ssize_t **PyDict_Size**(*PyObject* *p)

Part of the 안정 ABI. 딕셔너리에 있는 항목의 수를 반환합니다. 이는 딕셔너리에 대한 `len(p)`와 동등합니다.

int **PyDict_Next**(*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Part of the 안정 ABI. 딕셔너리 *p*의 모든 키-값 쌍을 이터레이트 합니다. *ppos*에 의해 참조된 *Py_ssize_t*는, 이터레이션을 시작하기 위해 이 함수를 처음 호출하기 전에 `0`으로 초기화되어야 합니다; 이 함수는 딕셔너리의 각 쌍에 대해 참을 반환하고, 모든 쌍이 보고되었으면 거짓을 반환합니다. 매개 변수 *pkey*와 *pvalue*는 각각 키와 값으로 채울 *PyObject** 변수를 가리 키거나, `NULL` 일 수 있습니다. 이들을 통해 반환된 참조는 모두 빌린(borrowed) 것입니다. 이터레이션 중에 *ppos*를 변경하면 안 됩니다. 이 값은 내부 딕셔너리 구조 내의 오프셋을 나타내며, 구조가 희박하므로 오프셋이 연속되지 않습니다.

예를 들면:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* 값으로 흥미로운 작업을 수행합니다... */
    ...
}
```

딕셔너리 *p*는 이터레이션 중에 변경해서는 안 됩니다. 딕셔너리를 이터레이트 할 때 값을 변경하는 것은 안전하지만, 키 집합이 변경되지 않는 한만 그렇습니다. 예를 들면:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

The function is not thread-safe in the *free-threaded* build without external synchronization. You can use *Py_BEGIN_CRITICAL_SECTION* to lock the dictionary while iterating over it:

```
Py_BEGIN_CRITICAL_SECTION(self->dict);
while (PyDict_Next(self->dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

참고

On the free-threaded build, this function can be used safely inside a critical section. However, the references returned for *pkey* and *pvalue* are *borrowed* and are only valid while the critical section is held. If you need to use these objects outside the critical section or when the critical section can be suspended, create a *strong reference* (for example, using *Py_NewRef()*).

int **PyDict_Merge**(*PyObject* *a, *PyObject* *b, int override)

Part of the 안정 ABI. 매핑 객체 *b*를 이터레이트 하면서, 키-값 쌍을 딕셔너리 *a*에 추가합니다. *b*는 딕셔너리거나 *PyMapping_Keys()*와 *PyObject_GetItem()*를 지원하는 모든 객체일 수 있습니다. *override*가 참이면, *a*에 있는 기존 쌍이 *b*에서 일치하는 키가 있으면 교체되고, 그렇지 않으면 *a*와 일치하는 키가 없을 때만 쌍이 추가됩니다. 성공하면 `0`을 반환하고, 예외가 발생하면 `-1`을 반환합니다.

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

Part of the 안정 ABI. 이는 C에서 `PyDict_Merge(a, b, 1)`와 같고, 두 번째 인자에 "keys" 어트리뷰트가 없을 때 *PyDict_Update()*가 키-값 쌍의 시퀀스에 대해 이터레이트 하지 않는다는 점만 제외하면, 파이썬에서 `a.update(b)`와 유사합니다. 성공하면 `0`을 반환하고, 예외가 발생하면 `-1`을 반환합니다.

int **PyDict_MergeFromSeq2**(*PyObject* *a, *PyObject* *seq2, int override)

Part of the 안정 ABI. *seq2*의 키-값 쌍으로 딕셔너리 *a*를 갱신하거나 병합합니다. *seq2*는 키-값 쌍으로 간주하는 길이 2의 이터러블 객체를 생성하는 이터러블 객체여야 합니다. 중복 키가 있으면, *override*가 참이면 마지막이 승리하고, 그렇지 않으면 첫 번째가 승리합니다. 성공 시 `0`을 반환하고, 예외가 발생하면 `-1`을 반환합니다. 동등한 파이썬은 이렇습니다(반환 값 제외)

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

int **PyDict_AddWatcher**(*PyDict_WatchCallback* callback)

Register *callback* as a dictionary watcher. Return a non-negative integer id which must be passed to future calls to *PyDict_Watch()*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyDict_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *PyDict_AddWatcher()*. Return `0` on success, `-1` on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

int **PyDict_Watch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as watched. The callback granted *watcher_id* by *PyDict_AddWatcher()* will be called when *dict* is modified or deallocated. Return `0` on success or `-1` on error.

Added in version 3.12.

int **PyDict_Unwatch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as no longer watched. The callback granted *watcher_id* by *PyDict_AddWatcher()* will no longer be called when *dict* is modified or deallocated. The dict must previously have been watched by this watcher. Return `0` on success or `-1` on error.

Added in version 3.12.

type **PyDict_WatchEvent**

Enumeration of possible dictionary watcher events: `PyDict_EVENT_ADDED`, `PyDict_EVENT_MODIFIED`, `PyDict_EVENT_DELETED`, `PyDict_EVENT_CLONED`, `PyDict_EVENT_CLEARED`, or `PyDict_EVENT_DEALLOCATED`.

Added in version 3.12.

typedef int (***PyDict_WatchCallback**)(*PyDict_WatchEvent* event, *PyObject* *dict, *PyObject* *key, *PyObject* *new_value)

Type of a dict watcher callback function.

If *event* is `PyDict_EVENT_CLEARED` or `PyDict_EVENT_DEALLOCATED`, both *key* and *new_value* will be `NULL`. If *event* is `PyDict_EVENT_ADDED` or `PyDict_EVENT_MODIFIED`, *new_value* will be the new value for *key*. If *event* is `PyDict_EVENT_DELETED`, *key* is being deleted from the dictionary and *new_value* will be `NULL`.

`PyDict_EVENT_CLONED` occurs when *dict* was previously empty and another dict is merged into it. To maintain efficiency of this operation, per-key `PyDict_EVENT_ADDED` events are not issued in this case; instead a single `PyDict_EVENT_CLONED` is issued, and *key* will be the source dictionary.

The callback may inspect but must not modify *dict*; doing so could have unpredictable effects, including infinite recursion. Do not trigger Python code execution in the callback, as it could modify the dict as a side effect.

If *event* is `PyDict_EVENT_DEALLOCATED`, taking a new reference in the callback to the about-to-be-destroyed dictionary will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Callbacks occur before the notified modification to *dict* takes place, so the prior state of *dict* can be inspected.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_WriteUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

9.4.2 집합 객체

이 절에서는 `set`과 `frozenset` 객체에 대한 공용 API에 대해 자세히 설명합니다. 아래 나열되지 않은 기능은 추상 객체 프로토콜 (*PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()* 및 *PyObject_GetIter()*를 포함합니다) 이나 추상 숫자 프로토콜 (*PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()* 및 *PyNumber_InPlaceXor()*을 포함합니다).

type **PySetObject**

이 *PyObject*의 서브 형은 `set`과 `frozenset` 객체 모두의 내부 데이터를 담는 데 사용됩니다. 이것은 작은 집합은 고정 크기(튜플 저장과 매우 흡사함)이고, 중형과 대형 집합은 별도의 가변 크기 메모리 블록(리스트 저장소처럼)을 가리킨다는 점에서 *PyDictObject*와 비슷합니다. 이 구조체의 필드는 아무것도 공개되지 않은 것으로 취급되어야 하며, 모두 변경될 수 있습니다. 모든 액세스는 구조체의 값을 조작하기보다는 설명된 API를 통해 수행해야 합니다.

PyTypeObject **PySet_Type**

Part of the 안정 ABI. 이것은 파이썬 `set` 형을 나타내는 *PyTypeObject*의 인스턴스입니다.

PyTypeObject **PyFrozenSet_Type**

Part of the 안정 ABI. 이것은 파이썬 `frozenset` 형을 나타내는 *PyTypeObject*의 인스턴스입니다.

다음 형 검사 매크로는 모든 파이썬 객체에 대한 포인터에서 작동합니다. 마찬가지로, 생성자 함수는 모든 이터러블 파이썬 객체에서 작동합니다.

int **PySet_Check**(*PyObject* *p)

*p*가 `set` 객체나 서브 형의 인스턴스면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyFrozenSet_Check**(*PyObject* *p)

*p*가 `frozenset` 객체나 서브 형의 인스턴스면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyAnySet_Check**(*PyObject* *p)

*p*가 `set` 객체, `frozenset` 객체 또는 서브 형의 인스턴스면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PySet_CheckExact**(*PyObject* *p)

*p*가 `set` 객체이지만, 서브 형의 인스턴스는 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

Added in version 3.10.

int **PyAnySet_CheckExact**(*PyObject* *p)

*p*가 `set` 객체나 `frozenset` 객체이지만, 서브 형의 인스턴스는 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyFrozenSet_CheckExact**(*PyObject* *p)

*p*가 `frozenset` 객체이지만, 서브 형의 인스턴스는 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PySet_New**(*PyObject* *iterable)

반환값: 새 참조. *Part of the* 안정 ABI. *iterable*에 의해 반환된 객체를 포함하는 새로운 `set`을 반환합니다. *iterable*은 새로운 빈 집합을 만들기 위해 `NULL` 일 수 있습니다. 성공하면 새 집합을, 실패하면 `NULL`을 반환합니다. *iterable*이 실제로 이터러블이 아니면 `TypeError`를 발생시킵니다. 생성자는 집합을 복사할 때도 유용합니다 (`c=set(s)`).

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

반환값: 새 참조. *Part of the* 안정 ABI. *iterable*에 의해 반환된 객체를 포함한 새로운 `frozenset`을 반환합니다. *iterable*은 새로운 빈 frozenset을 만들기 위해 `NULL` 일 수 있습니다. 성공하면 새 집합을, 실패하면 `NULL`을 반환합니다. *iterable*이 실제로 이터러블이 아니면 `TypeError`를 발생시킵니다.

`set` 이나 `frozenset`의 인스턴스 또는 그들의 서브 형의 인스턴스에 대해 다음 함수와 매크로를 사용할 수 있습니다.

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

Part of the 안정 ABI. `set` 이나 `frozenset` 객체의 길이를 반환합니다. `len(anyset)`와 동등합니다. *anyset*이 `set`, `frozenset` 또는 서브 형의 인스턴스가 아니면 `SystemError`를 발생시킵니다.

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)

에러 검사 없는 *PySet_Size()*의 매크로 형식.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

Part of the 안정 ABI. 발견되면 `1`을, 발견되지 않으면 `0`을, 에러가 발생하면 `-1`을 반환합니다. 파이썬 `__contains__()` 메서드와는 달리, 이 함수는 해시 불가능한 집합을 임시 frozenset으로 자동 변환하지 않습니다. *key*가 해시 불가능하면, `TypeError`를 발생시킵니다. *anyset*이 `set`, `frozenset` 또는 서브 형의 인스턴스가 아니면 `SystemError`를 발생시킵니다.

int **PySet_Add**(*PyObject* *set, *PyObject* *key)

Part of the 안정 ABI. *key*를 `set` 인스턴스에 추가합니다. 또한 `frozenset` 인스턴스에도 작동합니다 (*PyTuple_SetItem()*처럼 다른 코드에 노출되기 전에 새로운 frozenset의 값을 채우는 데 사용할 수 있습니다). 성공하면 `0`을, 실패하면 `-1`을 반환합니다. *key*가 해시 불가능하면, `TypeError`를 발생시킵니다. 성장할 공간이 없다면 `MemoryError`를 발생시킵니다. *set*이 `set` 이나 그 서브 형의 인스턴스가 아니면 `SystemError`를 발생시킵니다.

다음 함수는 `set` 이나 그것의 서브 형의 인스턴스에는 사용할 수 있지만, `frozenset` 이나 그 서브 형의 인스턴스에는 사용할 수 없습니다.

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

Part of the 안정 ABI. 발견되고 제거되면 `1`을 반환하고, 발견되지 않으면(아무런 일도 하지 않습니다) `0`을 반환하고, 에러가 발생하면 `-1`을 반환합니다. 발견할 수 없는 키에 대해 `KeyError`를 발생시키지 않습니다. *key*가 해시 불가능하면 `TypeError`를 발생시킵니다. 파이썬 `discard()` 메서드와는 달리, 이 함수는 해시 불가능한 집합을 임시 frozenset으로 자동 변환하지 않습니다. *set*이 `set` 이나 그 서브 형의 인스턴스가 아니면 `SystemError`를 발생시킵니다.

PyObject ***PySet_Pop**(*PyObject* *set)

반환값: 새 참조. *Part of the* 안정 ABI. *set*에 들어있는 임의의 객체에 대한 새 참조를 반환하고, *set*에서 객체를 제거합니다. 실패하면 `NULL`을 반환합니다. 집합이 비어 있으면, `KeyError`를 발생시킵니다. *set*이 `set` 이나 그 서브 형의 인스턴스가 아니면 `SystemError`를 발생시킵니다.

int **PySet_Clear**(*PyObject* *set)

Part of the 안정 ABI. 기존의 모든 요소 집합을 비웁니다. 성공하면 `0`을 반환합니다. *set*이 `set` 이나 그 서브타입의 인스턴스가 아니면 `SystemError`를 발생시키고 `-1`을 반환합니다.

9.5 함수 객체

9.5.1 함수 객체

파이썬 함수와 관련된 몇 가지 함수가 있습니다.

type **PyFunctionObject**

함수에 사용되는 C 구조체.

PyTypeObject **PyFunction_Type**

이것은 *PyTypeObject*의 인스턴스이며 파이썬 함수 형을 나타냅니다. 파이썬 프로그래머에게 `types.FunctionType`으로 노출됩니다.

int **PyFunction_Check**(*PyObject* *o)

*o*가 함수 객체(*PyFunction_Type* 형)면 참을 반환합니다. 매개 변수는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

반환값: 새 참조. 코드 객체 *code*와 연관된 새 함수 객체를 반환합니다. *globals*는 함수에서 액세스할 수 있는 전역 변수가 있는 딕셔너리이어야 합니다.

함수의 독스트링과 이름은 코드 객체에서 가져옵니다. `__module__`은 *globals*에서 가져옵니다. 인자 기본값, 어노테이션 및 클로저는 `NULL`로 설정됩니다. `__qualname__`은 코드 객체의 `co_qualname` 필드와 같은 값으로 설정됩니다.

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

반환값: 새 참조. *PyFunction_New()*와 비슷하지만, 함수 객체의 `__qualname__` 어트리뷰트를 설정할 수도 있도록 합니다. *qualname*는 유니코드 객체나 `NULL`이어야 합니다; `NULL`이면, `__qualname__` 어트리뷰트는 코드 객체의 `co_qualname` 필드와 같은 값으로 설정됩니다.

Added in version 3.3.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

반환값: 빌린 참조. 함수 객체 *op*와 연관된 코드 객체를 반환합니다.

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

반환값: 빌린 참조. 함수 객체 *op*와 연관된 전역 딕셔너리를 반환합니다.

PyObject ***PyFunction_GetModule**(*PyObject* *op)

반환값: 빌린 참조. 함수 객체 *op*의 `__module__` 어트리뷰트에 대한 빌린 참조를 반환합니다. *NULL* 일 수 있습니다.

이것은 일반적으로 모듈 이름을 포함하는 문자열이지만, 파이썬 코드로 다른 객체로 설정할 수 있습니다.

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

반환값: 빌린 참조. 함수 객체 *op*의 인자 기본값을 반환합니다. 이는 인자의 튜플이나 `NULL`일 수 있습니다.

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

함수 객체 *op*의 인자 기본값을 설정합니다. *defaults*는 `Py_None` 이나 튜플이어야 합니다.

실패하면 `SystemError`를 발생시키고 `-1`을 반환합니다.

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

Set the vectorcall field of a given function object *func*.

Warning: extensions using this API must preserve the behavior of the unaltered (default) vectorcall function!

Added in version 3.12.

PyObject ***PyFunction_GetKwDefaults**(*PyObject* *op)

반환값: 빌린 참조. Return the keyword-only argument default values of the function object *op*. This can be a dictionary of arguments or `NULL`.

PyObject ***PyFunction_GetClosure**(*PyObject* *op)

반환값: 빌린 참조. 함수 객체 *op*와 연관된 클로저를 반환합니다. 이것은 `NULL` 이나 셀 객체의 튜플일 수 있습니다.

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)

함수 객체 *op*와 연관된 클로저를 설정합니다. *closure*는 `Py_None` 이나 셀 객체의 튜플이어야 합니다.

실패하면 `SystemError`를 발생시키고 `-1`을 반환합니다.

PyObject ***PyFunction_GetAnnotations**(*PyObject* *op)

반환값: 빌린 참조. 함수 객체 *op*의 어노테이션을 반환합니다. 이것은 가변 딕셔너리나 `NULL` 일 수 있습니다.

int **PyFunction_SetAnnotations**(*PyObject* *op, *PyObject* *annotations)

함수 객체 *op*의 어노테이션을 설정합니다. *annotations*은 딕셔너리나 `Py_None` 이어야 합니다.

실패하면 `SystemError`를 발생시키고 `-1`을 반환합니다.

PyObject ***PyFunction_GET_CODE**(*PyObject* *op)

PyObject ***PyFunction_GET_GLOBALS**(*PyObject* *op)

PyObject ***PyFunction_GET_MODULE**(*PyObject* *op)

PyObject ***PyFunction_GET_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_KW_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_CLOSURE**(*PyObject* *op)

PyObject ***PyFunction_GET_ANNOTATIONS**(*PyObject* *op)

반환값: 빌린 참조. These functions are similar to their `PyFunction_Get*` counterparts, but do not do type checking. Passing anything other than an instance of *PyFunction_Type* is undefined behavior.

int **PyFunction_AddWatcher**(*PyFunction_WatchCallback* callback)

Register *callback* as a function watcher for the current interpreter. Return an ID which may be passed to `PyFunction_ClearWatcher()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyFunction_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from `PyFunction_AddWatcher()` for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type **PyFunction_WatchEvent**

Enumeration of possible function watcher events:

- `PyFunction_EVENT_CREATE`
- `PyFunction_EVENT_DESTROY`
- `PyFunction_EVENT_MODIFY_CODE`
- `PyFunction_EVENT_MODIFY_DEFAULTS`
- `PyFunction_EVENT_MODIFY_KWDEFAULTS`

Added in version 3.12.

typedef int (***PyFunction_WatchCallback**)(*PyFunction_WatchEvent* event, *PyFunctionObject* *func, *PyObject* *new_value)

Type of a function watcher callback function.

If *event* is `PyFunction_EVENT_CREATE` or `PyFunction_EVENT_DESTROY` then *new_value* will be `NULL`. Otherwise, *new_value* will hold a *borrowed reference* to the new value that is about to be stored in *func* for the attribute that is being modified.

The callback may inspect but must not modify *func*; doing so could have unpredictable effects, including infinite recursion.

If *event* is `PyFunction_EVENT_CREATE`, then the callback is invoked after *func* has been fully initialized. Otherwise, the callback is invoked before the modification to *func* takes place, so the prior state of *func* can be inspected. The runtime is permitted to optimize away the creation of function objects when possible. In such cases no event will be emitted. Although this creates the possibility of an observable difference of runtime behavior depending on optimization decisions, it does not change the semantics of the Python code being executed.

If *event* is `PyFunction_EVENT_DESTROY`, Taking a reference in the callback to the about-to-be-destroyed function will resurrect it, preventing it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using `PyErr_WriteUnraisable()`. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

9.5.2 인스턴스 메서드 객체

인스턴스 메서드는 `PyCFunction`에 대한 래퍼이며 `PyCFunction`를 클래스 객체에 연결하는 새로운 방법입니다. 이전의 `PyMethod_New(func, NULL, class)` 호출을 대체합니다.

PyTypeObject **PyInstanceMethod_Type**

이 *PyTypeObject* 인스턴스는 파이썬 인스턴스 메서드 형을 나타냅니다. 파이썬 프로그램에는 노출되지 않습니다.

int **PyInstanceMethod_Check**(*PyObject* *o)

*o*가 인스턴스 메서드 객체면 참을 반환합니다 (*PyInstanceMethod_Type* 형입니다). 매개 변수는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

반환값: 새 참조. 새 인스턴스 메서드 객체를 반환합니다. *func*는 임의의 콜러블 객체입니다. *func*는 인스턴스 메서드가 호출될 때 호출될 함수입니다.

PyObject ***PyInstanceMethod_Function**(*PyObject* *im)

반환값: 빌린 참조. 인스턴스 메서드 *im*과 연관된 함수 객체를 반환합니다.

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* *im)

반환값: 빌린 참조. 오류 검사를 피하는 *PyInstanceMethod_Function()*의 매크로 버전.

9.5.3 메서드 객체

메서드는 연결된(bound) 함수 객체입니다. 메서드는 항상 사용자 정의 클래스의 인스턴스에 연결됩니다. 연결되지 않은(unbound) 메서드(클래스 객체에 연결된 메서드)는 더는 사용할 수 없습니다.

PyTypeObject **PyMethod_Type**

이 *PyTypeObject* 인스턴스는 파이썬 메서드 형을 나타냅니다. 이것은 파이썬 프로그램에 types.MethodType로 노출됩니다.

int **PyMethod_Check**(*PyObject* *o)

*o*가 메서드 객체면 참을 반환합니다 (*PyMethod_Type* 형입니다). 매개 변수는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

PyObject ***PyMethod_New**(*PyObject* *func, *PyObject* *self)

반환값: 새 참조. 새로운 메서드 객체를 돌려줍니다. *func*는 임의의 콜러블 객체이며, *self*는 메서드가 연결되어야 할 인스턴스입니다. *func*는 메서드가 호출될 때 호출될 함수입니다. *self*는 NULL이 아니어야 합니다.

PyObject ***PyMethod_Function**(*PyObject* *meth)

반환값: 빌린 참조. *meth* 메서드와 연관된 함수 객체를 반환합니다.

PyObject ***PyMethod_GET_FUNCTION**(*PyObject* *meth)

반환값: 빌린 참조. 오류 검사를 피하는 *PyMethod_Function()*의 매크로 버전.

PyObject ***PyMethod_Self**(*PyObject* *meth)

반환값: 빌린 참조. *meth* 메서드와 연관된 인스턴스를 반환합니다.

PyObject ***PyMethod_GET_SELF**(*PyObject* *meth)

반환값: 빌린 참조. 오류 검사를 피하는 *PyMethod_Self()*의 매크로 버전.

9.5.4 셀 객체

"셀" 객체는 여러 스코프에서 참조하는 변수를 구현하는 데 사용됩니다. 이러한 변수마다, 값을 저장하기 위해 셀 객체가 만들어집니다; 값을 참조하는 각 스택 프레임의 지역 변수에는 해당 변수를 사용하는 외부 스코프의 셀에 대한 참조가 포함됩니다. 값에 액세스하면, 셀 객체 자체 대신 셀에 포함된 값이 사용됩니다. 이러한 셀 객체의 역참조(de-referencing)는 생성된 바이트 코드로부터의 지원이 필요합니다; 액세스 시 자동으로 역참조되지 않습니다. 셀 객체는 다른 곳에 유용하지는 않습니다.

type **PyCellObject**

셀 객체에 사용되는 C 구조체.

PyTypeObject **PyCell_Type**

셀 객체에 해당하는 형 객체.

int **PyCell_Check**(*PyObject* *ob)

*ob*가 셀 객체면 참을 반환합니다; *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

PyObject ***PyCell_New**(*PyObject* *ob)

반환값: 새 참조. *ob* 값을 포함하는 새 셀 객체를 만들고 반환합니다. 매개 변수는 NULL 일 수 있습니다.

PyObject ***PyCell_Get**(*PyObject* *cell)

반환값: 새 참조. 셀 *cell*의 내용을 반환합니다. *cell*은 NULL일 수 있습니다. *cell*이 셀 객체가 아니면, NULL을 반환하고 예외를 설정합니다.

PyObject ***PyCell_GET**(*PyObject* *cell)

반환값: 빌린 참조. 셀 *cell*의 내용을 반환하지만, *cell*이 NULL이 아닌지와 셀 객체인지를 확인하지 않습니다.

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)

셀 객체 *cell*의 내용을 *value*로 설정합니다. 이렇게 하면 셀의 현재 내용에 대한 참조를 해제합니다. *value*는 NULL 일 수 있습니다. *cell*는 NULL이 아니어야 합니다.

성공하면, 0을 반환합니다. *cell*이 셀 객체가 아니면, 예외를 설정하고 -1을 반환합니다.

void **PyCell_SET**(*PyObject* *cell, *PyObject* *value)

셀 객체 *cell*의 값을 *value*로 설정합니다. 참조 횟수는 조정되지 않고, 안전을 위한 검사가 이루어지지 않습니다; *cell*은 NULL이 아니어야 하고 셀 객체여야 합니다.

9.5.5 코드 객체

코드 객체는 CPython 구현의 저수준 세부 사항입니다. 각 객체는 아직 함수에 묶여 있지 않은 실행 가능한 코드 덩어리를 나타냅니다.

type **PyCodeObject**

코드 객체를 설명하는 데 사용되는 객체의 C 구조체. 이 형의 필드는 언제든지 변경될 수 있습니다.

PyTypeObject **PyCode_Type**

이것은 파이썬 코드 객체를 나타내는 *PyTypeObject*의 인스턴스입니다.

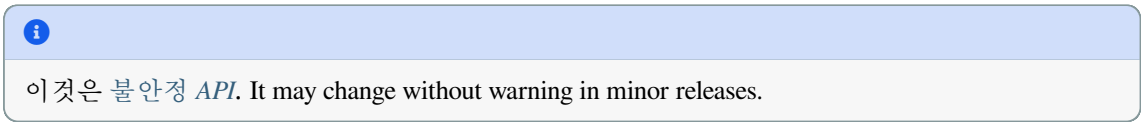
int **PyCode_Check**(*PyObject* *co)

*co*가 코드 객체면 참을 반환합니다. 이 함수는 항상 성공합니다.

Py_ssize_t **PyCode_GetNumFree**(*PyCodeObject* *co)

코드 객체에 있는 자유 (클로저) 변수의 개수를 반환합니다.

int **PyUnstable_Code_GetFirstFree**(*PyCodeObject* *co)



이것은 불안정 *API*. It may change without warning in minor releases.

Return the position of the first *free* (*closure*) *variable* in a code object.

버전 3.13에서 변경: Renamed from PyCode_GetFirstFree as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_New**(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

이것은 불안정 *API*. It may change without warning in minor releases.

새 코드 객체를 반환합니다. 프레임을 만들기 위해 더미 코드 객체가 필요하면, 대신 `PyCode_NewEmpty()`를 사용하십시오.

Since the definition of the bytecode changes often, calling `PyUnstable_Code_New()` directly can bind you to a precise Python version.

The many arguments of this function are inter-dependent in complex ways, meaning that subtle changes to values are likely to result in incorrect execution or VM crashes. Use this function only with extreme care.

버전 3.11에서 변경: Added `qualname` and `exceptiontable` parameters.

버전 3.12에서 변경: Renamed from `PyCode_New` as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_NewWithPosOnlyArgs**(int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

이것은 불안정 *API*. It may change without warning in minor releases.

`PyUnstable_Code_New()`와 비슷하지만, 위치 전용 인자를 위한 추가 "posonlyargcount"가 있습니다. `PyUnstable_Code_New`에 적용되는 것과 같은 주의 사항이 이 함수에도 적용됩니다.

Added in version 3.8: as `PyCode_NewWithPosOnlyArgs`

버전 3.11에서 변경: Added `qualname` and `exceptiontable` parameters.

버전 3.12에서 변경: Renamed to `PyUnstable_Code_NewWithPosOnlyArgs`. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyCode_NewEmpty**(const char *filename, const char *funcname, int firstlineno)

반환값: 새 참조. 지정된 파일명, 함수명 및 첫 번째 줄 번호를 갖는 새 빈 코드 객체를 반환합니다. 결과 코드 객체를 실행하면 `Exception`을 발생시킵니다.

int **PyCode_Addr2Line**(*PyCodeObject* *co, int byte_offset)

Return the line number of the instruction that occurs on or before `byte_offset` and ends after it. If you just need the line number of a frame, use `PyFrame_GetLineNumber()` instead.

For efficiently iterating over the line numbers in a code object, use **the API described in PEP 626**.

int **PyCode_Addr2Location**(*PyObject* *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

Sets the passed `int` pointers to the source code line and column numbers for the instruction at `byte_offset`. Sets the value to `0` when information is not available for any particular element.

Returns `1` if the function succeeds and 0 otherwise.

Added in version 3.11.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_code')`. Returns a strong reference to a

PyBytesObject representing the bytecode in a code object. On error, `NULL` is returned and an exception is raised.

This `PyBytesObject` may be created on-demand by the interpreter and does not necessarily represent the bytecode actually executed by CPython. The primary use case for this function is debuggers and profilers.

Added in version 3.11.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_varnames')`. Returns a new reference to a *PyTupleObject* containing the names of the local variables. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_cellvars')`. Returns a new reference to a *PyTupleObject* containing the names of the local variables that are referenced by nested functions. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

PyObject ***PyCode_GetFreevars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_freevars')`. Returns a new reference to a *PyTupleObject* containing the names of the *free (closure) variables*. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

int **PyCode_AddWatcher**(*PyCode_WatchCallback* callback)

Register *callback* as a code object watcher for the current interpreter. Return an ID which may be passed to *PyCode_ClearWatcher()*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyCode_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *PyCode_AddWatcher()* for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type **PyCodeEvent**

Enumeration of possible code object watcher events: - `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

Added in version 3.12.

typedef int (***PyCode_WatchCallback**)(*PyCodeEvent* event, *PyCodeObject* *co)

Type of a code object watcher callback function.

If *event* is `PY_CODE_EVENT_CREATE`, then the callback is invoked after *co* has been fully initialized. Otherwise, the callback is invoked before the destruction of *co* takes place, so the prior state of *co* can be inspected.

If *event* is `PY_CODE_EVENT_DESTROY`, taking a reference in the callback to the about-to-be-destroyed code object will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Users of this API should not rely on internal runtime implementation details. Such details may include, but are not limited to, the exact order and timing of creation and destruction of code objects. While changes in these details may result in differences observable by watchers (including whether a callback is invoked or not), it does not change the semantics of the Python code being executed.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_WriteUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

9.5.6 Code Object Flags

Code objects contain a bit-field of flags, which can be retrieved as the `co_flags` Python attribute (for example using *PyObject_GetAttrString()*), and set using a *flags* argument to *PyUnstable_Code_New()* and similar functions.

Flags whose names start with `CO_FUTURE_` correspond to features normally selectable by future statements. These flags can be used in *PyCompilerFlags.cf_flags*. Note that many `CO_FUTURE_` flags are mandatory in current versions of Python, and setting them has no effect.

The following flags are available. For their meaning, see the linked documentation of their Python equivalents.

Flag	Meaning
CO_OPTIMIZED	inspect.CO_OPTIMIZED
CO_NEWLOCALS	inspect.CO_NEWLOCALS
CO_VARARGS	inspect.CO_VARARGS
CO_VARKEYWORDS	inspect.CO_VARKEYWORDS
CO_NESTED	inspect.CO_NESTED
CO_GENERATOR	inspect.CO_GENERATOR
CO_COROUTINE	inspect.CO_COROUTINE
CO_ITERABLE_COROUTINE	inspect.CO_ITERABLE_COROUTINE
CO_ASYNC_GENERATOR	inspect.CO_ASYNC_GENERATOR
CO_HAS_DOCSTRING	inspect.CO_HAS_DOCSTRING
CO_METHOD	inspect.CO_METHOD
CO_FUTURE_DIVISION	no effect (__future__.division)
CO_FUTURE_ABSOLUTE_IMPORT	no effect (__future__.absolute_import)
CO_FUTURE_WITH_STATEMENT	no effect (__future__.with_statement)
CO_FUTURE_PRINT_FUNCTION	no effect (__future__.print_function)
CO_FUTURE_UNICODE_LITERALS	no effect (__future__.unicode_literals)
CO_FUTURE_GENERATOR_STOP	no effect (__future__.generator_stop)
CO_FUTURE_ANNOTATIONS	__future__.annotations

9.5.7 Extra information

To support low-level extensions to frame evaluation, such as external just-in-time compilers, it is possible to attach arbitrary extra data to code objects.

These functions are part of the unstable C API tier: this functionality is a CPython implementation detail, and the API may change without deprecation warnings.

Py_ssize_t **PyUnstable_Eval_RequestCodeExtraIndex**(*freefunc* free)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Return a new an opaque index value used to adding data to code objects.

You generally call this function once (per interpreter) and use the result with `PyCode_GetExtra` and `PyCode_SetExtra` to manipulate data on individual code objects.

If *free* is not `NULL`: when a code object is deallocated, *free* will be called on non-`NULL` data stored under the new index. Use *Py_DecRef()* when storing *PyObject*.

Added in version 3.6: as `_PyEval_RequestCodeExtraIndex`

버전 3.12에서 변경: Renamed to `PyUnstable_Eval_RequestCodeExtraIndex`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_GetExtra**(*PyObject* *code, *Py_ssize_t* index, void **extra)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Set *extra* to the extra data stored under the given index. Return 0 on success. Set an exception and return -1 on failure.

If no data was set under the index, set *extra* to `NULL` and return 0 without setting an exception.

Added in version 3.6: as `_PyCode_GetExtra`

버전 3.12에서 변경: Renamed to `PyUnstable_Code_GetExtra`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_SetExtra**(*PyObject* *code, *Py_ssize_t* index, void *extra)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.

Added in version 3.6: as `_PyCode_SetExtra`

버전 3.12에서 변경: Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

9.6 기타 객체

9.6.1 파일 객체

이 API는 C 표준 라이브러리의 버퍼링 된 I/O (`FILE*`) 지원에 의존하는 내장 파일 객체에 대한 파이썬 2 C API의 최소 에뮬레이션입니다. 파이썬 3에서, 파일과 스트림은 새로운 `io` 모듈을 사용합니다. 이 모듈은 운영 체제의 저수준 버퍼링 되지 않은 I/O 위에 여러 계층을 정의합니다. 아래에서 설명하는 함수는 이러한 새로운 API에 대한 편리한 C 래퍼이며, 주로 인터프리터의 내부 오류 보고를 위한 것입니다; 제삼자 코드는 대신 `io` API에 액세스하는 것이 좋습니다.

PyObject ***PyFile_FromFd**(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

반환값: 새 참조. *Part of the* 안정 ABI. 이미 열려있는 파일의 파일 기술자 *fd*로 파이썬 파일 객체를 만듭니다. 인자 *name*, *encoding*, *errors* 및 *newline*은 기본값을 사용하기 위해 `NULL` 일 수 있습니다; *buffering*은 기본값을 사용하기 위해 *-1* 일 수 있습니다. *name*은 무시되고, 이전 버전과의 호환성을 위해 유지됩니다. 실패 시 `NULL`을 반환합니다. 인자에 대한 더 자세한 설명은 `io.open()` 함수 설명서를 참조하십시오.

> **경고**
>
> 파이썬 스트림이 자체적인 버퍼링 계층을 가지고 있으므로, OS 수준의 파일 기술자와 혼합하면 여러 예기치 못한 문제가 발생할 수 있습니다 (가령 데이터의 예상치 못한 순서).

버전 3.2에서 변경: *name* 어트리뷰트를 무시합니다.

int **PyObject_AsFileDescriptor**(*PyObject* *p)

Part of the 안정 ABI. *p*와 관련된 파일 기술자를 `int`로 반환합니다. 객체가 정수면, 값이 반환됩니다. 그렇지 않으면 객체의 `fileno()` 메서드가 있으면 호출됩니다; 메서드는 반드시 정수를 반환해야 하고, 그 값이 파일 기술자 값으로 반환됩니다. 실패하면 예외를 설정하고 `-1`을 반환합니다.

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

반환값: 새 참조. *Part of the* 안정 ABI. `p.readline([n])`과 동등합니다. 이 함수는 객체 *p*에서 한 줄을 읽습니다. *p*는 파일 객체나 `readline()` 메서드가 있는 임의의 객체일 수 있습니다. *n*이 `0`이면, 줄의 길이와 관계없이 정확히 한 줄을 읽습니다. *n*이 `0`보다 크면, *n* 바이트 이상을 파일에서 읽지 않습니다; 불완전한 줄이 반환될 수 있습니다. 두 경우 모두, 파일 끝에 즉시 도달하면 빈 문자열이 반환됩니다. 그러나 *n*이 `0`보다 작으면, 길이와 관계없이 한 줄을 읽지만, 파일 끝에 즉시 도달하면 `EOFError`가 발생합니다.

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

제공된 handler를 통해 매개 변수를 전달하도록 `io.open_code()`의 일반적인 동작을 재정의합니다.

*handler*는 다음과 같은 형의 함수입니다:

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

`PyObject *(*)(PyObject *path, void *userData)`와 동등합니다. 여기서 *path*는 `PyUnicodeObject`임이 보장됩니다.

userData 포인터는 훅 함수로 전달됩니다. 훅 함수는 다른 런타임에서 호출될 수 있으므로, 이 포인터는 파이썬 상태를 직접 참조하면 안 됩니다.

이 훅은 의도적으로 임포트 중에 사용되므로, 고정되었거나(frozen) `sys.modules`에 있다고 알려진 경우가 아니라면 훅 실행 중에 새로운 모듈을 임포트하는 것을 피하십시오.

일단 훅이 설정되면, 제거하거나 교체할 수 없으며, 이후의 `PyFile_SetOpenCodeHook()`에 대한 호출은 실패합니다. 실패 시, 함수는 -1을 반환하고 인터프리터가 초기화되었으면 예외를 설정합니다.

이 함수는 `Py_Initialize()` 전에 호출해도 안전합니다.

인자 없이 감사 이벤트 `setopencodehook`을 발생시킵니다.

Added in version 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

Part of the 안정 ABI. 객체 *obj*를 파일 객체 *p*에 씁니다. *flags*에서 지원되는 유일한 플래그는 *Py_PRINT_RAW*입니다; 주어지면, `repr()` 대신 객체의 `str()`이 기록됩니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다; 적절한 예외가 설정됩니다.

int **PyFile_WriteString**(const char *s, *PyObject* *p)

Part of the 안정 ABI. 문자열 *s*를 파일 객체 *p*에 씁니다. 성공하면 `0`을 반환하고, 실패하면 `-1`을 반환합니다; 적절한 예외가 설정됩니다.

9.6.2 모듈 객체

PyTypeObject **PyModule_Type**

Part of the 안정 ABI. 이 *PyTypeObject* 인스턴스는 파이썬 모듈 형을 나타냅니다. 이것은 `types.ModuleType`으로 파이썬 프로그램에 노출됩니다.

int **PyModule_Check**(*PyObject* *p)

*p*가 모듈 객체이거나 모듈 객체의 서브 형이면 참을 반환합니다. 이 함수는 항상 성공합니다.

int **PyModule_CheckExact**(*PyObject* *p)

*p*가 모듈 객체이지만, *PyModule_Type*의 서브 형이 아니면 참을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyModule_NewObject**(*PyObject* *name)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. `module.__name__`이 *name*으로 설정된 새 모듈 객체를 반환합니다. 모듈의 `__name__`, `__doc__`, `__package__` 및 `__loader__` 어트리뷰트가 채워집니다 (`__name__`을 제외하고 모두 `None`으로 설정됩니다). `__file__` 어트리뷰트를 설정하는 것은 호출자의 책임입니다.

Return `NULL` with an exception set on error.

Added in version 3.3.

버전 3.4에서 변경: `__package__`와 `__loader__`가 이제 `None`으로 설정됩니다.

PyObject ***PyModule_New**(const char *name)

반환값: 새 참조. *Part of the* 안정 ABI. *PyModule_NewObject()*와 비슷하지만, name이 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

PyObject ***PyModule_GetDict**(*PyObject* *module)

반환값: 빌린 참조. *Part of the* 안정 ABI. *module*의 이름 공간을 구현하는 딕셔너리 객체를 반환합니다; 이 객체는 모듈 객체의 `__dict__` 어트리뷰트와 같습니다. *module*이 모듈 객체(또는 모듈 객체의 서브 형)가 아니면, `SystemError`가 발생하고 `NULL`이 반환됩니다.

확장은 모듈의 `__dict__`를 직접 조작하지 말고 다른 `PyModule_*`과 `PyObject_*` 함수를 사용하는 것이 좋습니다.

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. *module*의 `__name__` 값을 반환합니다. 모듈이 제공하지 않거나, 문자열이 아니면, `SystemError`가 발생하고 `NULL`이 반환됩니다.

Added in version 3.3.

const char ***PyModule_GetName**(*PyObject* *module)

Part of the 안정 ABI. *PyModule_GetNameObject()*와 비슷하지만 `'utf-8'`로 인코딩된 이름을 반환합니다.

void ***PyModule_GetState**(*PyObject* *module)

Part of the 안정 ABI. 모듈의 "상태", 즉 모듈 생성 시 할당된 메모리 블록을 가리키는 포인터나 `NULL`을 반환합니다. *PyModuleDef.m_size*를 참조하십시오.

PyModuleDef ***PyModule_GetDef**(*PyObject* *module)

Part of the 안정 ABI. 모듈이 만들어진 *PyModuleDef* 구조체에 대한 포인터나 모듈이 정의에서 만들어지지 않았으면 `NULL`을 반환합니다.

PyObject ***PyModule_GetFilenameObject**(*PyObject* *module)

반환값: 새 참조. *Part of the* 안정 ABI. *module*의 `__file__` 어트리뷰트를 사용하여 *module*이 로드된 파일 이름을 반환합니다. 정의되지 않았거나 문자열이 아니면, `SystemError`를 발생시키고 `NULL`을 반환합니다; 그렇지 않으면 유니코드 객체에 대한 참조를 반환합니다.

Added in version 3.2.

const char ***PyModule_GetFilename**(*PyObject* *module)

Part of the 안정 ABI. *PyModule_GetFilenameObject()*와 비슷하지만 'utf-8'로 인코딩된 파일명을 반환합니다.

버전 3.2부터 폐지됨: *PyModule_GetFilename()*은 인코딩할 수 없는 파일명에 대해 `UnicodeEncodeError`를 발생시킵니다, 대신 *PyModule_GetFilenameObject()*를 사용하십시오.

9.6.3 Module definitions

The functions in the previous section work on any module object, including modules imported from Python code.

Modules defined using the C API typically use a *module definition*, *PyModuleDef* – a statically allocated, constant "description" of how a module should be created.

The definition is usually used to define an extension's "main" module object (see *Defining extension modules* for details). It is also used to *create extension modules dynamically*.

Unlike *PyModule_New()*, the definition allows management of *module state* – a piece of memory that is allocated and cleared together with the module object. Unlike the module's Python attributes, Python code cannot replace or delete data stored in module state.

type **PyModuleDef**

Part of the 안정 ABI (*including all members*). The module definition struct, which holds all information needed to create a module object. This structure must be statically allocated (or be otherwise guaranteed to be valid while any modules created from it exist). Usually, there is only one variable of this type for each extension module.

PyModuleDef_Base **m_base**

이 멤버를 항상 `PyModuleDef_HEAD_INIT`로 초기화하십시오.

const char ***m_name**

새 모듈의 이름.

const char ***m_doc**

모듈의 독스트링; 일반적으로 *PyDoc_STRVAR*로 만들어진 독스트링 변수가 사용됩니다.

Py_ssize_t **m_size**

모듈 상태는 정적 전역이 아닌 *PyModule_GetState()*로 조회할 수 있는 모듈별 메모리 영역에 유지될 수 있습니다. 이것은 여러 서브 인터프리터에서 모듈을 사용하는 것을 안전하게 만듭니다.

이 메모리 영역은 모듈 생성 시 *m_size*를 기준으로 할당되며, 모듈 객체가 할당 해제될 때 (있다면 *m_free* 함수가 호출된 후에) 해제됩니다.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state. Negative `m_size` is only allowed when using *legacy single-phase initialization* or when *creating modules dynamically*.

자세한 내용은 **PEP 3121**을 참조하십시오.

PyMethodDef ***m_methods**

PyMethodDef 값으로 기술되는 모듈 수준 함수 테이블에 대한 포인터. 함수가 없으면 `NULL`일 수 있습니다.

PyModuleDef_Slot ***m_slots***

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using legacy single-phase initialization, *m_slots* must be `NULL`.

버전 3.5에서 변경: 버전 3.5 이전에는, 이 멤버가 항상 `NULL`로 설정되었으며, 다음과 같이 정의되었습니다:

inquiry **m_reload**

traverseproc **m_traverse**

모듈 객체의 GC 탐색 중 호출할 탐색 함수나, 필요하지 않으면 `NULL`.

모듈 상태가 요청되었지만, 아직 할당되지 않았으면 이 함수가 호출되지 않습니다. 이것은 모듈이 만들이진 직후, 모듈이 실행되기 직전의 경우입니다 (*Py_mod_exec* 함수). 더 정확하게는, *m_size*가 0보다 크고 모듈 상태 (*PyModule_GetState()*가 반환하는) 가 `NULL`이면 이 함수가 호출되지 않습니다.

버전 3.9에서 변경: 모듈 상태가 할당되기 전에 더는 호출되지 않습니다.

inquiry **m_clear**

모듈 객체의 GC 정리 중에 호출할 정리(clear) 함수나, 필요하지 않으면 `NULL`.

모듈 상태가 요청되었지만, 아직 할당되지 않았으면 이 함수가 호출되지 않습니다. 이것은 모듈이 만들이진 직후, 모듈이 실행되기 직전의 경우입니다 (*Py_mod_exec* 함수). 더 정확하게는, *m_size*가 0보다 크고 모듈 상태 (*PyModule_GetState()*가 반환하는) 가 `NULL`이면 이 함수가 호출되지 않습니다.

Like *PyTypeObject.tp_clear*, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and *m_free* is called directly.

버전 3.9에서 변경: 모듈 상태가 할당되기 전에 더는 호출되지 않습니다.

freefunc **m_free**

모듈 객체 할당 해제 중에 호출할 함수나, 필요하지 않으면 `NULL`.

모듈 상태가 요청되었지만, 아직 할당되지 않았으면 이 함수가 호출되지 않습니다. 이것은 모듈이 만들이진 직후, 모듈이 실행되기 직전의 경우입니다 (*Py_mod_exec* 함수). 더 정확하게는, *m_size*가 0보다 크고 모듈 상태 (*PyModule_GetState()*가 반환하는) 가 `NULL`이면 이 함수가 호출되지 않습니다.

버전 3.9에서 변경: 모듈 상태가 할당되기 전에 더는 호출되지 않습니다.

Module slots

type **PyModuleDef_Slot**

int **slot**

아래 설명된 사용 가능한 값 중에서 선택된, 슬롯 ID.

void ***value**

슬롯 ID에 따라 그 의미가 달라지는, 슬롯의 값.

Added in version 3.5.

사용 가능한 슬롯 형은 다음과 같습니다:

Py_mod_create

모듈 객체 자체를 만들기 위해 호출되는 함수를 지정합니다. 이 슬롯의 *value* 포인터는 다음과 같은 서명을 갖는 함수를 가리켜야 합니다:

PyObject ***create_module**(*PyObject* *spec, *PyModuleDef* *def)

이 함수는 **PEP 451**에 정의된 대로, `ModuleSpec` 인스턴스와 모듈 정의를 받습니다. 새 모듈 객체를 반환하거나, 에러를 설정하고 `NULL`을 반환해야 합니다.

이 함수는 최소한으로 유지해야 합니다. 특히 같은 모듈을 다시 임포트 하려고 시도하면 무한 루프가 발생할 수 있어서, 임의의 파이썬 코드를 호출하면 안 됩니다.

하나의 모듈 정의에서 여러 `Py_mod_create` 슬롯을 지정할 수 없습니다.

`Py_mod_create`를 지정하지 않으면, 임포트 절차는 *PyModule_New()*를 사용하여 일반 모듈 객체를 만듭니다. 이름은 정의가 아니라 *spec*에서 취합니다, 확장 모듈이 단일 모듈 정의를 공유하면서 모듈 계층 구조에서 해당 위치에 동적으로 조정되고 심볼릭 링크를 통해 다른 이름으로 임포트 될 수 있도록 하기 위함입니다.

반환된 객체가 *PyModule_Type*의 인스턴스 일 필요는 없습니다. 임포트 관련 어트리뷰트 설정과 읽기를 지원하는 한 모든 형을 사용할 수 있습니다. 그러나, `PyModuleDef`에 `NULL`이 아닌 `m_traverse`, `m_clear`, `m_free`; 0이 아닌 `m_size`; 또는 `Py_mod_create` 이외의 슬롯이 있으면, `PyModule_Type` 인스턴스 만 반환될 수 있습니다.

Py_mod_exec

모듈을 실행하기 위해 호출되는 함수를 지정합니다. 이것은 파이썬 모듈의 코드를 실행하는 것과 동등합니다: 일반적으로, 이 함수는 클래스와 상수를 모듈에 추가합니다. 함수의 서명은 다음과 같습니다:

int **exec_module**(*PyObject* *module)

여러 개의 `Py_mod_exec` 슬롯이 지정되면, *m_slots* 배열에 나타나는 순서대로 처리됩니다.

Py_mod_multiple_interpreters

Specifies one of the following values:

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

The module does not support being imported in subinterpreters.

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

The module supports being imported in subinterpreters, but only when they share the main interpreter's GIL. (See isolating-extensions-howto.)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

The module supports being imported in subinterpreters, even when they have their own GIL. (See isolating-extensions-howto.)

This slot determines whether or not importing this module in a subinterpreter will fail.

하나의 모듈 정의에서 여러 `Py_mod_multiple_interpreters` 슬롯을 지정할 수 없습니다.

If `Py_mod_multiple_interpreters` is not specified, the import machinery defaults to `Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED`.

Added in version 3.12.

Py_mod_gil

Specifies one of the following values:

Py_MOD_GIL_USED

The module depends on the presence of the global interpreter lock (GIL), and may access global state without synchronization.

Py_MOD_GIL_NOT_USED

The module is safe to run without an active GIL.

This slot is ignored by Python builds not configured with `--disable-gil`. Otherwise, it determines whether or not importing this module will cause the GIL to be automatically enabled. See whatsnew313-free-threaded-cpython for more detail.

하나의 모듈 정의에서 여러 `Py_mod_gil` 슬롯을 지정할 수 없습니다.

If `Py_mod_gil` is not specified, the import machinery defaults to `Py_MOD_GIL_USED`.

Added in version 3.13.

9.6.4 Creating extension modules dynamically

The following functions may be used to create a module outside of an extension's *initialization function*. They are also used in *single-phase initialization*.

PyObject ***PyModule_Create**(*PyModuleDef* *def)

반환값: 새 참조. Create a new module object, given the definition in *def*. This is a macro that calls `PyModule_Create2()` with *module_api_version* set to `PYTHON_API_VERSION`, or to `PYTHON_ABI_VERSION` if using the *limited API*.

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

반환값: 새 참조. *Part of the* 안정 ABI. *def*의 정의에 따라, API 버전 *module_api_version*을 가정하여 새 모듈 객체를 만듭니다. 해당 버전이 실행 중인 인터프리터 버전과 일치하지 않으면, `RuntimeWarning`을 발생시킵니다.

Return `NULL` with an exception set on error.

This function does not support slots. The `m_slots` member of *def* must be `NULL`.

> **참고**
>
> 이 함수는 대부분 `PyModule_Create()`를 대신 사용해야 합니다; 확실히 필요할 때만 사용하십시오.

PyObject ***PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)

반환값: 새 참조. This macro calls `PyModule_FromDefAndSpec2()` with *module_api_version* set to `PYTHON_API_VERSION`, or to `PYTHON_ABI_VERSION` if using the *limited API*.

Added in version 3.5.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. API 버전 *module_api_version*을 가정하여, 주어진 *def*의 정의와 ModuleSpec *spec*으로 새 모듈 객체를 만듭니다. 해당 버전이 실행 중인 인터프리터 버전과 일치하지 않으면, `RuntimeWarning`을 발생시킵니다.

Return `NULL` with an exception set on error.

Note that this does not process execution slots (`Py_mod_exec`). Both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

> **참고**
>
> 이 함수는 대부분 `PyModule_FromDefAndSpec()`을 대신 사용해야 합니다; 확실히 필요할 때만 사용하십시오.

Added in version 3.5.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

Part of the 안정 ABI 버전 *3.7* 이후로. *def*에 지정된 모든 실행 슬롯(`Py_mod_exec`)을 처리합니다.

Added in version 3.5.

PYTHON_API_VERSION

The C API version. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

PYTHON_ABI_VERSION

Defined as `3` for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

9.6.5 지원 함수

The following functions are provided to help initialize a module state. They are intended for a module' s execution slots (*Py_mod_exec*), the initialization function for legacy *single-phase initialization*, or code that creates modules dynamically.

int **PyModule_AddObjectRef**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the 안정 ABI 버전 *3.10* 이후로. *module*에 객체를 *name*으로 추가합니다. 모듈의 초기화 함수에서 사용할 수 있는 편의 함수입니다.

성공하면, `0`을 반환합니다. 에러 시, 예외를 발생시키고 `-1`을 반환합니다.

사용 예:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_DECREF(obj);
    return res;
}
```

To be convenient, the function accepts `NULL` *value* with an exception set. In this case, return `-1` and just leave the raised exception unchanged.

The example can also be written without checking explicitly if *obj* is `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
}
```

Note that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `NULL`.

The number of different *name* strings passed to this function should be kept small, usually by only using statically allocated strings as *name*. For names that aren' t known at compile time, prefer calling *PyUnicode_FromString()* and *PyObject_SetAttr()* directly. For more details, see *PyUnicode_InternFromString()*, which may be used internally to create a key object.

Added in version 3.10.

int **PyModule_Add**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the 안정 ABI 버전 *3.13* 이후로. Similar to *PyModule_AddObjectRef()*, but "steals" a reference to *value*. It can be called with a result of function that returns a new reference without bothering to check its result or even saving it to a variable.

사용 예:

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value)) < 0) {
    goto error;
}
```

Added in version 3.13.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the 안정 ABI. Similar to `PyModule_AddObjectRef()`, but steals a reference to *value* on success (if it returns `0`).

The new `PyModule_Add()` or `PyModule_AddObjectRef()` functions are recommended, since it is easy to introduce reference leaks by misusing the `PyModule_AddObject()` function.

참고

참조를 훔치는 다른 함수와 달리, `PyModule_AddObject()`는 **성공 시**에만 *value*에 대한 참조를 해제합니다.

이는 반환 값을 확인해야 하며, 에러 시 호출하는 코드가 수동으로 *value*를 `Py_XDECREF()` 해야 함을 뜻합니다.

사용 예:

```
PyObject *obj = PyBytes_FromString(value);
if (PyModule_AddObject(module, "spam", obj) < 0) {
    // 'obj' 가 NULL 이 아니고 PyModule_AddObject() 가 실패하면,
    // 'obj' 강한 참조는 Py_XDECREF() 로 삭제해야 합니다.
    // 'obj' 가 NULL 이면, Py_XDECREF() 는 아무것도 하지 않습니다.
    Py_XDECREF(obj);
    goto error;
}
// PyModule_AddObject() 는 obj 에 대한 참조를 훔칩니다:
// 여기에서 Py_XDECREF(obj) 는 필요 없습니다.
```

버전 3.13부터 폐지됨: `PyModule_AddObject()` is *soft deprecated*.

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)

Part of the 안정 ABI. *module*에 정수 상수를 *name*으로 추가합니다. 이 편의 함수는 모듈의 초기화 함수에서 사용할 수 있습니다. 에러 시 예외를 설정하고 `-1`을, 성공하면 `0`을 반환합니다.

This is a convenience function that calls `PyLong_FromLong()` and `PyModule_AddObjectRef()`; see their documentation for details.

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

Part of the 안정 ABI. *module*에 문자열 상수를 *name*으로 추가합니다. 이 편의 함수는 모듈의 초기화 함수에서 사용할 수 있습니다. 문자열 *value*는 `NULL`로 끝나야 합니다. 에러 시 예외를 설정하고 `-1`을, 성공 시 `0`을 반환합니다.

This is a convenience function that calls `PyUnicode_InternFromString()` and `PyModule_AddObjectRef()`; see their documentation for details.

PyModule_AddIntMacro(module, macro)

*module*에 int 상수를 추가합니다. 이름과 값은 *macro*에서 취합니다. 예를 들어 `PyModule_AddIntMacro(module, AF_INET)`은 *AF_INET* 값을 가진 int 상수 *AF_INET*을 *module*에 추가합니다. 에러 시 예외를 설정하고 `-1`을, 성공하면 `0`을 반환합니다.

PyModule_AddStringMacro(module, macro)

*module*에 문자열 상수를 추가합니다.

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

Part of the 안정 ABI 버전 *3.10* 이후로. *module*에 형 객체를 추가합니다. 내부적으로 `PyType_Ready()`를 호출하여 형 객체를 파이널라이즈합니다. 형 객체의 이름은 점 뒤 `tp_name`의 마지막 구성 요소에서 가져옵니다. 에러가 발생하면 예외를 설정하고 `-1`을, 성공하면 `0`을 반환합니다.

Added in version 3.9.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)

Part of the 안정 ABI 버전 *3.7* 이후로. Add the functions from the `NULL` terminated *functions* array to *module*. Refer to the `PyMethodDef` documentation for details on individual entries (due to the lack of a shared module namespace, module level "functions" implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes).

This function is called automatically when creating a module from `PyModuleDef` (such as when using *Multi-phase initialization*, `PyModule_Create`, or `PyModule_FromDefAndSpec`). Some module authors may prefer defining functions in multiple `PyMethodDef` arrays; in that case they should call this function directly.

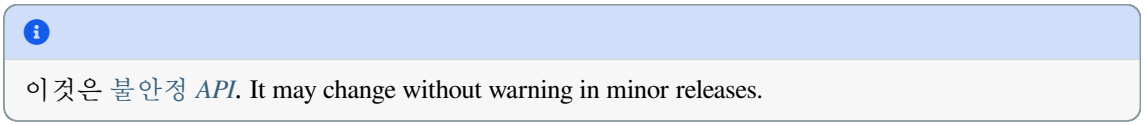
Added in version 3.5.

int **PyModule_SetDocString**(*PyObject* *module, const char *docstring)

Part of the 안정 ABI 버전 *3.7* 이후로. Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef` (such as when using *Multi-phase initialization*, `PyModule_Create`, or `PyModule_FromDefAndSpec`).

Added in version 3.5.

int **PyUnstable_Module_SetGIL**(*PyObject* *module, void *gil)



이것은 불안정 *API*. It may change without warning in minor releases.

Indicate that *module* does or does not support running without the global interpreter lock (GIL), using one of the values from `Py_mod_gil`. It must be called during *module*' s initialization function when using *Legacy single-phase initialization*. If this function is not called during module initialization, the import machinery assumes the module does not support running without the GIL. This function is only available in Python builds configured with `--disable-gil`. Return `-1` with an exception set on error, `0` on success.

Added in version 3.13.

Module lookup (single-phase initialization)

The legacy *single-phase initialization* initialization scheme creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

이 함수들은 다단계 초기화를 사용하여 만들어진 모듈에서는 작동하지 않습니다. 단일 정의에서 그러한 모듈이 여러 개 만들어질 수 있기 때문입니다.

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

반환값: 빌린 참조. *Part of the* 안정 ABI. 현재 인터프리터에 대해 *def*에서 만들어진 모듈 객체를 반환합니다. 이 메서드를 사용하려면 먼저 모듈 객체가 `PyState_AddModule()`로 인터프리터 상태에 연결되어 있어야 합니다. 해당 모듈 객체를 찾을 수 없거나 인터프리터 상태에 아직 연결되지 않았으면, `NULL`을 반환합니다.

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

Part of the 안정 ABI 버전 *3.3* 이후로. 함수에 전달된 모듈 객체를 인터프리터 상태에 연결합니다. 이는 `PyState_FindModule()`을 통해 모듈 객체에 액세스 할 수 있도록 합니다.

단단계 초기화를 사용하여 만든 모듈에만 효과가 있습니다.

Python calls `PyState_AddModule` automatically after importing a module that uses *single-phase initialization*, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only

if the module' s own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

If a module was attached previously using the same *def*, it is replaced by the new *module*.

The caller must have an *attached thread state*.

에러 시 예외를 설정하고 `-1`을, 성공 시 `0`을 반환합니다.

Added in version 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

Part of the 안정 ABI 버전 *3.3* 이후로. *def*에서 만들어진 모듈 객체를 인터프리터 상태에서 제거합니다. 에러 시 예외를 설정하고 `-1`을, 성공 시 `0`을 반환합니다.

The caller must have an *attached thread state*.

Added in version 3.3.

9.6.6 이터레이터 객체

파이썬은 두 개의 범용 이터레이터 객체를 제공합니다. 첫째, 시퀀스 이터레이터는 `__getitem__()` 메서드를 지원하는 임의의 시퀀스와 작동합니다. 둘째는 콜러블 객체와 종료 신호(sentinel) 값을 사용하고, 시퀀스의 각 항목에 대해 콜러블을 호출하고, 종료 신호 값이 반환될 때 이터레이션을 종료합니다.

PyTypeObject **PySeqIter_Type**

Part of the 안정 ABI. *PySeqIter_New()*와 내장 시퀀스 형에 대한 `iter()` 내장 함수의 단일 인자 형식에 의해 반환된 이터레이터 객체에 대한 형 객체.

int **PySeqIter_Check**(*PyObject* *op)

*op*의 형이 *PySeqIter_Type*이면 참을 돌려줍니다. 이 함수는 항상 성공합니다.

PyObject ***PySeqIter_New**(*PyObject* *seq)

반환값: 새 참조. *Part of the* 안정 ABI. 일반 시퀀스 객체 *seq*와 함께 작동하는 이터레이터를 반환합니다. 시퀀스가 서브스크립션 연산에서 `IndexError`를 일으키면 이터레이션이 끝납니다.

PyTypeObject **PyCallIter_Type**

Part of the 안정 ABI. *PyCallIter_New()*와 `iter()` 내장 함수의 두 인자 형식에 의해 반환된 이터레이터 객체에 대한 형 객체.

int **PyCallIter_Check**(*PyObject* *op)

*op*의 형이 *PyCallIter_Type*이면 참을 돌려줍니다. 이 함수는 항상 성공합니다.

PyObject ***PyCallIter_New**(*PyObject* *callable, *PyObject* *sentinel)

반환값: 새 참조. *Part of the* 안정 ABI. 새로운 이터레이터를 돌려줍니다. 첫 번째 매개 변수 *callable*은 매개 변수 없이 호출할 수 있는 모든 파이썬 콜러블 객체일 수 있습니다; 각 호출은 이터레이션의 다음 항목을 반환해야 합니다. *callable*이 *sentinel*와 같은 값을 반환하면 이터레이션이 종료됩니다.

9.6.7 디스크립터 객체

"디스크립터"는 객체의 일부 어트리뷰트를 기술하는 객체입니다. 그것들은 형 객체의 딕셔너리에 있습니다.

PyTypeObject **PyProperty_Type**

Part of the 안정 ABI. 내장 디스크립터 형들을 위한 형 객체.

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

반환값: 새 참조. *Part of the* 안정 ABI.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

반환값: 새 참조. *Part of the* 안정 ABI.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

반환값: 새 참조. *Part of the* 안정 ABI.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

반환값: 새 참조.

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

반환값: 새 참조. *Part of the* 안정 ABI.

int **PyDescr_IsData**(*PyObject* *descr)

디스크립터 객체 *descr*가 데이터 어트리뷰트를 기술하고 있으면 0이 아닌 값을 반환하고, 메서드를 기술하면 `0`을 돌려줍니다. *descr*는 디스크립터 객체여야 합니다; 오류 검사는 없습니다.

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

반환값: 새 참조. *Part of the* 안정 ABI.

9.6.8 슬라이스 객체

PyTypeObject **PySlice_Type**

Part of the 안정 ABI. 슬라이스 객체의 형 객체. 이것은 파이썬 계층의 `slice`와 같습니다.

int **PySlice_Check**(*PyObject* *ob)

*ob*가 슬라이스 객체면 참을 반환합니다. *ob*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

반환값: 새 참조. *Part of the* 안정 ABI. 지정된 값으로 새로운 슬라이스 객체를 반환합니다. *start*, *stop* 및 *step* 매개 변수는 같은 이름의 슬라이스 객체 어트리뷰트의 값으로 사용됩니다. 모든 값은 `NULL` 일 수 있으며, 이 경우 `None`이 해당 어트리뷰트에 사용됩니다.

새 객체를 할당할 수 없으면 예외를 설정하고 `NULL`을 반환합니다.

int **PySlice_GetIndices**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Part of the 안정 ABI. 길이가 *length*인 시퀀스를 가정하여, 슬라이스 객체 *slice*에서 start, stop 및 step 인덱스를 가져옵니다. *length*보다 큰 인덱스를 에러로 처리합니다.

성공하면 `0`을 반환하고, 에러면 예외 설정 없이 `-1`을 반환합니다 (인덱스 중 하나가 `None`이 아니고 정수로 변환되지 않는 한, 이때는 예외를 설정하고 `-1`을 반환합니다).

이 기능을 사용하고 싶지는 않을 것입니다.

버전 3.2에서 변경: 전에는 *slice* 매개 변수의 매개 변수 형이 `PySliceObject*`였습니다.

int **PySlice_GetIndicesEx**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step, *Py_ssize_t* *slicelength)

Part of the 안정 ABI. *PySlice_GetIndices()*를 쓸만하게 대체합니다. 길이가 *length*인 시퀀스를 가정하여, 슬라이스 객체 *slice*에서 start, stop 및 step 인덱스를 가져오고, *slicelength*에 슬라이스의 길이를 저장합니다. 범위를 벗어난 인덱스는 일반 슬라이스의 처리와 일관된 방식으로 잘립니다.

성공하면 `0`을 반환하고, 에러면 예외를 설정하고 `-1`을 반환합니다.

참고

이 함수는 크기를 조정할 수 있는 시퀀스에는 안전하지 않은 것으로 간주합니다. 호출은 *PySlice_Unpack()*와 *PySlice_AdjustIndices()*의 조합으로 대체되어야 합니다. 즉

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength)
↪< 0) {
    // 에러 반환
}
```

은 다음으로 대체됩니다

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // 에러 반환
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

버전 3.2에서 변경: 전에는 *slice* 매개 변수의 매개 변수 형이 `PySliceObject*`였습니다.

버전 3.6.1에서 변경: `Py_LIMITED_API`가 설정되어 있지 않거나 `0x03050400`과 `0x03060000` (포함하지 않음) 사이나 `0x03060100` 이상의 값으로 설정되었으면, `PySlice_GetIndicesEx()`는 `PySlice_Unpack()`과 `PySlice_AdjustIndices()`를 사용하는 매크로로 구현됩니다. 인자 *start*, *stop* 및 *step*는 여러 번 평가됩니다.

버전 3.6.1부터 폐지됨: `Py_LIMITED_API`가 `0x03050400`보다 작거나 `0x03060000`과 `0x03060100` (포함하지 않음) 사이의 값으로 설정되었으면 `PySlice_GetIndicesEx()`는 폐지된 함수입니다.

int **PySlice_Unpack**(*PyObject* *slice, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Part of the 안정 ABI 버전 *3.7* 이후로. 슬라이스 객체의 start, stop 및 step 데이터 멤버를 C 정수로 추출합니다. `PY_SSIZE_T_MAX`보다 큰 값을 `PY_SSIZE_T_MAX`로 조용히 줄이고, `PY_SSIZE_T_MIN`보다 작은 start 와 stop 값을 `PY_SSIZE_T_MIN`로 조용히 높이고, `-PY_SSIZE_T_MAX`보다 작은 step 값을 `-PY_SSIZE_T_MAX`로 조용히 높입니다.

에러면 예외를 설정하고 `-1`을, 성공하면 `0`을 반환합니다.

Added in version 3.6.1.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

Part of the 안정 ABI 버전 *3.7* 이후로. 지정된 length 길이의 시퀀스를 가정하여 start/stop 슬라이스 인덱스를 조정합니다. 범위를 벗어난 인덱스는 일반 슬라이스의 처리와 일관된 방식으로 잘립니다.

슬라이스의 길이를 반환합니다. 항상 성공합니다. 파이썬 코드를 호출하지 않습니다.

Added in version 3.6.1.

Ellipsis 객체

PyTypeObject **PyEllipsis_Type**

Part of the 안정 ABI. 파이썬 `Ellipsis` 객체의 형. 파이썬 계층의 `types.EllipsisType`과 같습니다.

PyObject ***Py_Ellipsis**

파이썬 `Ellipsis` 객체. 이 객체에는 메서드가 없습니다. *Py_None*과 마찬가지로, 불멸 싱글톤 객체입니다.

버전 3.12에서 변경: *Py_Ellipsis*는 불멸입니다.

9.6.9 MemoryView 객체

`memoryview` 객체는 C 수준 버퍼 인터페이스를 다른 객체와 마찬가지로 전달될 수 있는 파이썬 객체로 노출합니다.

PyObject ***PyMemoryView_FromObject**(*PyObject* *obj)

반환값: 새 참조. *Part of the* 안정 ABI. 버퍼 인터페이스를 제공하는 객체에서 memoryview 객체를 만듭니다. *obj*가 쓰기 가능한 버퍼 제공을 지원하면, memoryview 객체는 읽기/쓰기가 되고, 그렇지 않으면 읽기 전용이거나 제공자의 재량에 따라 읽기/쓰기가 될 수 있습니다.

PyBUF_READ

읽기 전용 버퍼를 요청하는 플래그

PyBUF_WRITE

쓰기 가능한 버퍼를 요청하는 플래그

PyObject ***PyMemoryView_FromMemory**(char *mem, *Py_ssize_t* size, int flags)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.7* 이후로. *mem*를 하부 버퍼로 사용하여 memoryview 객체를 만듭니다. *flags*는 `PyBUF_READ` 나 `PyBUF_WRITE` 중 하나일 수 있습니다.

Added in version 3.3.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.11* 이후로. 주어진 버퍼 구조체 *view*를 감싸는 memoryview 객체를 만듭니다. 간단한 바이트 버퍼의 경우는, `PyMemoryView_FromMemory()`가 선호되는 함수입니다.

PyObject ***PyMemoryView_GetContiguous**(*PyObject* *obj, int buffertype, char order)

반환값: 새 참조. *Part of the* 안정 ABI. 버퍼 인터페이스를 정의하는 객체로부터 메모리의 연속 청크('C' 나 'F' ortran *order*로)로 memoryview 객체를 만듭니다. 메모리가 연속적이면 memoryview 객체는 원래 메모리를 가리킵니다. 그렇지 않으면, 복사본이 만들어지고 memoryview는 새 바이트열 객체를 가리킵니다.

*buffertype*은 `PyBUF_READ` 나 `PyBUF_WRITE` 중 하나일 수 있습니다.

int **PyMemoryView_Check**(*PyObject* *obj)

객체 *obj*가 memoryview 객체면 참을 반환합니다. 현재는 `memoryview`의 서브 클래스를 만들 수 없습니다. 이 함수는 항상 성공합니다.

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* *mview)

제공자 버퍼의 memoryview의 비공개 복사본의 포인터를 돌려줍니다. *mview*는 **반드시** memoryview 인스턴스여야 합니다; 이 매크로는 형을 확인하지 않으므로 직접 검사해야 합니다, 그렇지 않으면 충돌 위험이 있습니다.

PyObject ***PyMemoryView_GET_BASE**(*PyObject* *mview)

memoryview가 기반으로 하는 제공자 객체에 대한 포인터나 memoryview가 `PyMemoryView_FromMemory()` 나 `PyMemoryView_FromBuffer()` 함수 중 하나로 만들어졌으면 `NULL`을 반환합니다. *mview*는 **반드시** memoryview 인스턴스여야 합니다.

9.6.10 약한 참조 객체

파이썬은 약한 참조를 1급 객체로 지원합니다. 약한 참조를 직접 구현하는 두 가지 구체적인 객체 형이 있습니다. 첫 번째는 간단한 참조 객체이며, 두 번째는 가능한 한 원래 객체의 프락시 역할을 합니다.

int **PyWeakref_Check**(*PyObject* *ob)

*ob*가 참조 객체나 프락시 객체면 0이 아닌 값을 반환합니다. 이 함수는 항상 성공합니다.

int **PyWeakref_CheckRef**(*PyObject* *ob)

*ob*가 참조 객체면 0이 아닌 값을 반환합니다. 이 함수는 항상 성공합니다.

int **PyWeakref_CheckProxy**(*PyObject* *ob)

*ob*가 프락시 객체면 0이 아닌 값을 반환합니다. 이 함수는 항상 성공합니다.

PyObject ***PyWeakref_NewRef**(*PyObject* *ob, *PyObject* *callback)

반환값: 새 참조. *Part of the* 안정 ABI. *ob* 객체에 대한 약한 참조 객체를 반환합니다. 이것은 항상 새로운 참조를 돌려주지만, 새로운 객체를 생성하는 것이 보장되지는 않습니다; 기존 참조 객체가 반환될 수 있습니다. 두 번째 매개 변수인 *callback*은 *ob*가 가비지 수집될 때 알림을 받는 콜러블 객체가 될 수 있습니다; 하나의 매개 변수를 받아들여야 하는데, 약한 참조 객체 자체입니다. *callback*은 `None` 이나 `NULL` 일 수도 있습니다. *ob*가 약하게 참조할 수 있는 객체가 아니거나, *callback*이 콜러블, `None` 또는 `NULL`이 아니면, `NULL`을 반환하고 `TypeError`를 발생시킵니다.

PyObject ***PyWeakref_NewProxy**(*PyObject* *ob, *PyObject* *callback)

반환값: 새 참조. *Part of the* 안정 ABI. *ob* 객체에 대한 약한 참조 프락시 객체를 반환합니다. 이것은 항상 새로운 참조를 돌려주지만, 새로운 객체를 생성하는 것이 보장되지는 않습니다; 기존 프락시 객체가 반환될 수 있습니다. 두 번째 매개 변수인 *callback*은 *ob*가 가비지 수집될 때 알림을 받는 콜러블 객체가 될 수 있습니다; 하나의 매개 변수를 받아들여야 하는데, 약한 참조 객체 자체입니다. *callback*은 `None` 이나 `NULL` 일 수도 있습니다. *ob*가 약하게 참조할 수 있는 객체가 아니거나, *callback*이 콜러블, `None` 또는 `NULL`이 아니면, `NULL`을 반환하고 `TypeError`를 발생시킵니다.

int **PyWeakref_GetRef**(*PyObject* *ref, *PyObject* **pobj)

Part of the 안정 ABI 버전 *3.13* 이후로. Get a *strong reference* to the referenced object from a weak reference, *ref*, into **pobj*.

- On success, set **pobj* to a new *strong reference* to the referenced object and return 1.
- If the reference is dead, set **pobj* to `NULL` and return 0.
- On error, raise an exception and return -1.

Added in version 3.13.

PyObject ***PyWeakref_GetObject**(*PyObject* *ref)

반환값: 빌린 참조. *Part of the* 안정 ABI. 약한 참조(*ref*)로부터 참조된 객체에 대한 빌린 참조를 반환합니다. 참조가 더는 살아있지 않으면, `Py_None`을 반환합니다.

> **참고**
>
> 이 함수는 참조된 객체에 대한 빌린 참조를 반환합니다. 이는 빌린 참조의 마지막 사용 전에 파괴될 수 없을 때를 제외하고, 객체에 대해 항상 *Py_INCREF()*를 호출해야 함을 뜻합니다.

Deprecated since version 3.13, will be removed in version 3.15: 대신 *PyWeakref_GetRef()*를 사용하십시오.

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* *ref)

반환값: 빌린 참조. *PyWeakref_GetObject()*와 유사하지만, 에러 검사를 수행하지 않습니다.

Deprecated since version 3.13, will be removed in version 3.15: 대신 *PyWeakref_GetRef()*를 사용하십시오.

int **PyWeakref_IsDead**(*PyObject* *ref)

Test if the weak reference *ref* is dead. Returns 1 if the reference is dead, 0 if it is alive, and -1 with an error set if *ref* is not a weak reference object.

Added in version 3.14.

void **PyObject_ClearWeakRefs**(*PyObject* *object)

Part of the 안정 ABI. This function is called by the *tp_dealloc* handler to clear weak references.

This iterates through the weak references for *object* and calls callbacks for those references which have one. It returns when all callbacks have been attempted.

void **PyUnstable_Object_ClearWeakRefsNoCallbacks**(*PyObject* *object)

> 이것은 불안정 *API*. It may change without warning in minor releases.

Clears the weakrefs for *object* without calling the callbacks.

This function is called by the *tp_dealloc* handler for types with finalizers (i.e., `__del__()`). The handler for those objects first calls *PyObject_ClearWeakRefs()* to clear weakrefs and call their callbacks, then the finalizer, and finally this function to clear any weakrefs that may have been created by the finalizer.

In most circumstances, it' s more appropriate to use *PyObject_ClearWeakRefs()* to clear weakrefs instead of this function.

Added in version 3.13.

9.6.11 캡슐

이 객체 사용에 대한 자세한 정보는 using-capsules를 참조하십시오.

Added in version 3.1.

type **PyCapsule**

이 *PyObject*의 서브 형은 불투명한 값을 나타내며, 파이썬 코드를 통해 다른 C 코드로 불투명한 값(`void*` 포인터로)을 전달해야 하는 C 확장 모듈에 유용합니다. 이것은 한 모듈에서 정의된 C 함수 포인터를 다른 모듈에서 사용할 수 있게 만드는 데 종종 사용되므로, 일반 임포트 메커니즘을 사용하여 동적으로 로드된 모듈에 정의된 C API에 액세스할 수 있습니다.

type **PyCapsule_Destructor**

Part of the 안정 ABI. 캡슐에 대한 파괴자(destructor) 콜백 형. 이렇게 정의됩니다:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

PyCapsule_Destructor 콜백의 의미는 *PyCapsule_New()*를 참조하십시오.

int **PyCapsule_CheckExact**(*PyObject* *p)

인자가 *PyCapsule*이면 참을 돌려줍니다. 이 함수는 항상 성공합니다.

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

반환값: 새 참조. *Part of the* 안정 ABI. *pointer*를 캡슐화하는 *PyCapsule*을 만듭니다. *pointer* 인자는 `NULL`이 아닐 수도 있습니다.

실패하면, 예외를 설정하고 `NULL`을 반환합니다.

name 문자열은 `NULL` 이나 유효한 C 문자열에 대한 포인터일 수 있습니다. `NULL`이 아니면, 이 문자열은 캡슐보다 오래 유지되어야 합니다. (*destructor* 내부에서 해제할 수는 있습니다.)

destructor 인자가 `NULL`이 아니면, 캡슐이 파괴될 때 캡슐을 인자로 호출됩니다.

이 캡슐을 모듈의 어트리뷰트로 저장하려면, *name*을 `modulename.attributename`로 지정해야 합니다. 이렇게 하면 다른 모듈이 *PyCapsule_Import()*를 사용하여 캡슐을 임포트 할 수 있습니다.

void ***PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)

Part of the 안정 ABI. 캡슐에 저장된 *pointer*를 가져옵니다. 실패하면, 예외를 설정하고 `NULL`을 반환합니다.

name 매개 변수는 캡슐에 저장된 이름과 정확하게 비교되어야 합니다. 캡슐에 저장된 이름이 `NULL`이면, 전달된 *name*도 `NULL` 이어야 합니다. 파이썬은 C 함수 `strcmp()`를 사용하여 캡슐 이름을 비교합니다.

PyCapsule_Destructor ***PyCapsule_GetDestructor**(*PyObject* *capsule)

Part of the 안정 ABI. 캡슐에 저장된 현재 파괴자를 반환합니다. 실패하면, 예외를 설정하고 `NULL`을 반환합니다.

캡슐이 `NULL` 파괴자를 갖는 것은 합법적입니다. 이것은 `NULL` 반환 코드를 다소 모호하게 만듭니다; 명확히 하려면 *PyCapsule_IsValid()* 나 *PyErr_Occurred()*를 사용하십시오.

void ***PyCapsule_GetContext**(*PyObject* *capsule)

Part of the 안정 ABI. 캡슐에 저장된 현재 컨텍스트를 반환합니다. 실패하면, 예외를 설정하고 `NULL`을 반환합니다.

캡슐이 `NULL` 컨텍스트를 갖는 것은 합법적입니다. 이것은 `NULL` 반환 코드를 다소 모호하게 만듭니다; 명확히 하려면 *PyCapsule_IsValid()* 나 *PyErr_Occurred()*를 사용하십시오.

const char ***PyCapsule_GetName**(*PyObject* *capsule)

Part of the 안정 ABI. 캡슐에 저장된 현재 이름을 반환합니다. 실패하면, 예외를 설정하고 `NULL`을 반환합니다.

캡슐이 `NULL` 이름을 갖는 것은 합법적입니다. 이것은 `NULL` 반환 코드를 다소 모호하게 만듭니다; 명확히 하려면 *PyCapsule_IsValid()* 나 *PyErr_Occurred()*를 사용하십시오.

void *PyCapsule_Import(const char *name, int no_block)

Part of the 안정 ABI. 모듈의 캡슐 어트리뷰트에서 C 객체에 대한 포인터를 임포트 합니다. *name* 매개변수는 `module.attribute` 처럼 어트리뷰트의 전체 이름을 지정해야 합니다. 캡슐에 저장된 *name* 은, 이 문자열과 정확히 일치해야 합니다.

This function splits *name* on the `.` character, and imports the first element. It then processes further elements using attribute lookups.

성공하면 캡슐의 내부 *pointer*를 반환합니다. 실패하면, 예외를 설정하고 `NULL`를 반환합니다.

> **참고**
>
> If *name* points to an attribute of some submodule or subpackage, this submodule or subpackage must be previously imported using other means (for example, by using *PyImport_ImportModule()*) for the attribute lookups to succeed.

버전 3.3에서 변경: *no_block*은 더는 효과가 없습니다.

int PyCapsule_IsValid(*PyObject* *capsule, const char *name)

Part of the 안정 ABI. *capsule*이 유효한 캡슐인지를 판단합니다. 유효한 캡슐은 `NULL`이 아니며, *PyCapsule_CheckExact()*를 통과하고, `NULL`이 아닌 포인터가 저장되며, 내부 이름이 *name* 매개변수와 일치합니다. (캡슐 이름을 비교하는 방법에 대한 정보는 *PyCapsule_GetPointer()*를 참조하십시오.)

즉, *PyCapsule_IsValid()*가 참값을 반환하면, 모든 접근자(`PyCapsule_Get`으로 시작하는 모든 함수)에 대한 호출이 성공함이 보장됩니다.

객체가 유효하고 전달된 이름과 일치하면 0이 아닌 값을 반환합니다. 그렇지 않으면 `0`을 반환합니다. 이 함수는 실패하지 않습니다.

int PyCapsule_SetContext(*PyObject* *capsule, void *context)

Part of the 안정 ABI. *capsule* 내부의 컨텍스트 포인터를 *context*로 설정합니다.

성공하면 `0`을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

int PyCapsule_SetDestructor(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

Part of the 안정 ABI. *capsule* 내부의 파괴자를 *destructor*로 설정합니다.

성공하면 `0`을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

int PyCapsule_SetName(*PyObject* *capsule, const char *name)

Part of the 안정 ABI. *capsule* 내부의 이름을 *name*으로 설정합니다. `NULL`이 아니면, 이름은 캡슐보다 오래 유지되어야 합니다. 캡슐에 저장된 이전 *name*이 `NULL`이 아니면, 이를 해제하려고 시도하지 않습니다.

성공하면 `0`을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

int PyCapsule_SetPointer(*PyObject* *capsule, void *pointer)

Part of the 안정 ABI. *capsule* 내부의 void 포인터를 *pointer*로 설정합니다. 포인터는 `NULL`이 아닐 수 있습니다.

성공하면 `0`을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

9.6.12 Frame Objects

type PyFrameObject

Part of the 제한된 API (*as an opaque struct*). The C structure of the objects used to describe frame objects.

There are no public members in this structure.

버전 3.11에서 변경: The members of this structure were removed from the public C API. Refer to the What's New entry for details.

The *PyEval_GetFrame()* and *PyThreadState_GetFrame()* functions can be used to get a frame object.

See also *Reflection*.

PyTypeObject **PyFrame_Type**

The type of frame objects. It is the same object as `types.FrameType` in the Python layer.

버전 3.11에서 변경: Previously, this type was only available after including `<frameobject.h>`.

int **PyFrame_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame object.

버전 3.11에서 변경: Previously, this function was only available after including `<frameobject.h>`.

PyFrameObject ***PyFrame_GetBack**(*PyFrameObject* *frame)

반환값: 새 참조. Get the *frame* next outer frame.

Return a *strong reference*, or `NULL` if *frame* has no outer frame.

Added in version 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

반환값: 새 참조. Get the *frame*' s `f_builtins` attribute.

Return a *strong reference*. The result cannot be `NULL`.

Added in version 3.11.

PyCodeObject ***PyFrame_GetCode**(*PyFrameObject* *frame)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.10* 이후로. Get the *frame* code.

Return a *strong reference*.

The result (frame code) cannot be `NULL`.

Added in version 3.9.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

반환값: 새 참조. Get the generator, coroutine, or async generator that owns this frame, or `NULL` if this frame is not owned by a generator. Does not raise an exception, even if the return value is `NULL`.

Return a *strong reference*, or `NULL`.

Added in version 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

반환값: 새 참조. Get the *frame*' s `f_globals` attribute.

Return a *strong reference*. The result cannot be `NULL`.

Added in version 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

Get the *frame*' s `f_lasti` attribute.

Returns -1 if `frame.f_lasti` is `None`.

Added in version 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

반환값: 새 참조. Get the variable *name* of *frame*.

- Return a *strong reference* to the variable value on success.
- Raise `NameError` and return `NULL` if the variable does not exist.
- Raise an exception and return `NULL` on error.

name type must be a `str`.

Added in version 3.12.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

반환값: 새 참조. Similar to `PyFrame_GetVar()`, but the variable name is a C string encoded in UTF-8.

Added in version 3.12.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

반환값: 새 참조. Get the *frame*'s `f_locals` attribute. If the frame refers to an *optimized scope*, this returns a write-through proxy object that allows modifying the locals. In all other cases (classes, modules, `exec()`, `eval()`) it returns the mapping representing the frame locals directly (as described for `locals()`).

Return a *strong reference*.

Added in version 3.11.

버전 3.13에서 변경: As part of **PEP 667**, return an instance of `PyFrameLocalsProxy_Type`.

int **PyFrame_GetLineNumber**(*PyFrameObject* *frame)

Part of the 안정 ABI *버전 3.10 이후로.* Return the line number that *frame* is currently executing.

Frame Locals Proxies

Added in version 3.13.

The `f_locals` attribute on a frame object is an instance of a "frame-locals proxy". The proxy object exposes a write-through view of the underlying locals dictionary for the frame. This ensures that the variables exposed by `f_locals` are always up to date with the live local variables in the frame itself.

See **PEP 667** for more information.

PyTypeObject **PyFrameLocalsProxy_Type**

The type of frame `locals()` proxy objects.

int **PyFrameLocalsProxy_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame `locals()` proxy.

Internal Frames

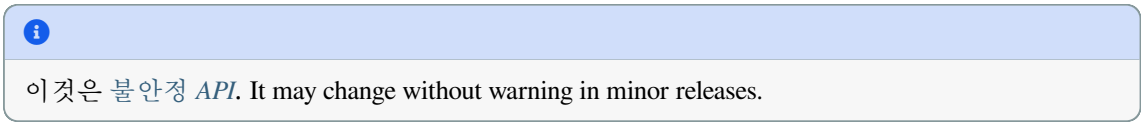
Unless using **PEP 523**, you will not need this.

struct **_PyInterpreterFrame**

The interpreter's internal frame representation.

Added in version 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

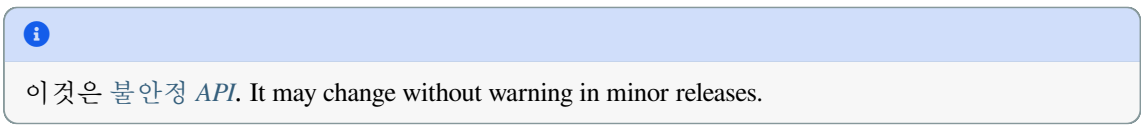


이것은 불안정 *API*. It may change without warning in minor releases.

Return a *strong reference* to the code object for the frame.

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

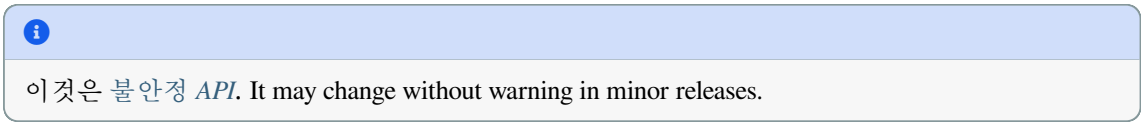


이것은 불안정 *API*. It may change without warning in minor releases.

Return the byte offset into the last executed instruction.

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);



이것은 불안정 *API*. It may change without warning in minor releases.

Return the currently executing line number, or -1 if there is no line number.

Added in version 3.12.

9.6.13 제너레이터 객체

제너레이터 객체는 파이썬이 제너레이터 이터레이터를 구현하기 위해 사용하는 객체입니다. 일반적으로 `PyGen_New()` 또는 `PyGen_NewWithQualName()`를 명시적으로 호출하는 것이 아니라, 값을 일드(yield)하는 함수를 이터레이트하여 만들어집니다.

type **PyGenObject**

제너레이터 객체에 사용되는 C 구조체.

PyTypeObject **PyGen_Type**

제너레이터 객체에 해당하는 형 객체

int **PyGen_Check**(*PyObject* *ob)

*ob*가 제너레이터 객체면 참을 돌려줍니다; *ob*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyGen_CheckExact**(*PyObject* *ob)

*ob*의 형이 `PyGen_Type`이면 참을 돌려줍니다; *ob*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

PyObject ***PyGen_New**(*PyFrameObject* *frame)

반환값: 새 참조. *frame* 객체에 기반한 새 제너레이터 객체를 만들어 반환합니다. 이 함수는 *frame*에 대한 참조를 훔칩니다. 인자는 `NULL`이 아니어야 합니다.

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

반환값: 새 참조. *frame* 객체에 기반한 새 제너레이터 객체를 만들어 반환하는데, `__name__` 과 `__qualname__`를 *name* 및 *qualname*로 설정합니다. 이 함수는 *frame*에 대한 참조를 훔칩니다. *frame* 인자는 `NULL`이 아니어야 합니다.

9.6.14 코루틴 객체

Added in version 3.5.

코루틴 객체는 `async` 키워드로 선언된 함수가 반환하는 것입니다.

type **PyCoroObject**

코루틴 객체에 사용되는 C 구조체.

PyTypeObject **PyCoro_Type**

코루틴 객체에 해당하는 형 객체.

int **PyCoro_CheckExact**(*PyObject* *ob)

*ob*의 형이 `PyCoro_Type`이면 참을 반환합니다. *ob*는 `NULL`일 수 없습니다. 이 함수는 항상 성공합니다.

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

반환값: 새 참조. *frame* 객체를 기반으로 새 코루틴 객체를 만들어서 반환합니다. `__name__` 과 `__qualname__`은 *name* 과 *qualname*로 설정합니다. 이 함수는 *frame*에 대한 참조를 훔칩니다. *frame* 인자는 `NULL`일 수 없습니다.

9.6.15 컨텍스트 변수 객체

Added in version 3.7.

버전 3.7.1에서 변경:

참고

파이썬 3.7.1에서 모든 컨텍스트 변수 C API의 서명이 *PyContext*, *PyContextVar* 및 *PyContextToken* 대신 *PyObject* 포인터를 사용하도록 **변경되었습니다**, 예를 들어:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

자세한 내용은 bpo-34762를 참조하십시오.

이 절에서는 `contextvars` 모듈을 위한 공용 C API에 대해 자세히 설명합니다.

type **PyContext**

`contextvars.Context` 객체를 나타내는 데 사용되는 C 구조체.

type **PyContextVar**

`contextvars.ContextVar` 객체를 나타내는 데 사용되는 C 구조체.

type **PyContextToken**

`contextvars.Token` 객체를 나타내는 데 사용되는 C 구조체.

PyTypeObject **PyContext_Type**

context 형을 나타내는 형 객체.

PyTypeObject **PyContextVar_Type**

컨텍스트 변수 형을 나타내는 형 객체.

PyTypeObject **PyContextToken_Type**

컨텍스트 변수 토큰 형을 나타내는 형 객체.

형 검사 매크로:

int **PyContext_CheckExact**(*PyObject* *o)

*o*가 *PyContext_Type* 형이면 참을 돌려줍니다. *o*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyContextVar_CheckExact**(*PyObject* *o)

*o*가 *PyContextVar_Type* 형이면 참을 돌려줍니다. *o*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyContextToken_CheckExact**(*PyObject* *o)

*o*가 *PyContextToken_Type* 형이면 참을 돌려줍니다. *o*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

컨텍스트 객체 관리 함수:

PyObject ***PyContext_New**(void)

반환값: 새 참조. 새로운 빈 컨텍스트 객체를 만듭니다. 에러가 발생하면 `NULL`를 반환합니다.

PyObject ***PyContext_Copy**(*PyObject* *ctx)

반환값: 새 참조. 전달된 *ctx* 컨텍스트 객체의 얕은 복사본을 만듭니다. 에러가 발생하면 `NULL`을 반환합니다.

PyObject ***PyContext_CopyCurrent**(void)

반환값: 새 참조. 현재 스레드 컨텍스트의 얕은 복사본을 만듭니다. 에러가 발생하면 `NULL`을 반환합니다.

int **PyContext_Enter**(*PyObject* *ctx)

현재 스레드의 현재 컨텍스트로 *ctx*를 설정합니다. 성공 시 `0`을 반환하고, 에러 시 `-1`을 반환합니다.

int **PyContext_Exit**(*PyObject* *ctx)

ctx 컨텍스트를 비활성화하고 이전 컨텍스트를 현재 스레드의 현재 컨텍스트로 복원합니다. 성공 시 `0`을 반환하고, 에러 시 `-1`을 반환합니다.

int **PyContext_AddWatcher**(*PyContext_WatchCallback* callback)

Register *callback* as a context object watcher for the current interpreter. Return an ID which may be passed to *PyContext_ClearWatcher()*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.14.

int **PyContext_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *PyContext_AddWatcher()* for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.14.

type **PyContextEvent**

Enumeration of possible context object watcher events:

- `Py_CONTEXT_SWITCHED`: The *current context* has switched to a different context. The object passed to the watch callback is the now-current `contextvars.Context` object, or None if no context is current.

Added in version 3.14.

typedef int (***PyContext_WatchCallback**)(*PyContextEvent* event, *PyObject* *obj)

Context object watcher callback function. The object passed to the callback is event-specific; see *PyContextEvent* for details.

If the callback returns with an exception set, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_FormatUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.14.

컨텍스트 변수 함수:

PyObject ***PyContextVar_New**(const char *name, *PyObject* *def)

반환값: 새 참조. 새 `ContextVar` 객체를 만듭니다. *name* 매개 변수는 인트로스펙션과 디버그 목적으로 사용됩니다. *def* 매개 변수는 컨텍스트 변수의 기본값을 지정할 수 있습니다. `NULL`은 기본값 없음입니다. 에러가 발생하면, 이 함수는 `NULL`을 반환합니다.

int **PyContextVar_Get**(*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

컨텍스트 변수의 값을 가져옵니다. 조회하는 동안 에러가 발생하면 `-1`을 반환하고, 값이 있는지와 상관없이 에러가 발생하지 않으면 `0`을 반환합니다.

컨텍스트 변수가 발견되면, *value*는 그것을 가리키는 포인터가 됩니다. 컨텍스트 변수가 발견되지 않으면, *value*는 다음을 가리 킵니다:

- *default_value*, `NULL`이 아니면;
- *var*의 기본값, `NULL`이 아니면;
- `NULL`

`NULL`일 때를 제외하고, 이 함수는 새 참조를 반환합니다.

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)

반환값: 새 참조. 현재 컨텍스트에서 *var*의 값을 *value*로 설정합니다. 이 변경에 대한 새 토큰 객체를 반환하거나, 에러가 발생하면 `NULL`을 반환합니다.

int **PyContextVar_Reset**(*PyObject* *var, *PyObject* *token)

var 컨텍스트 변수의 상태를 *token*을 반환한 *PyContextVar_Set()* 호출 전의 상태로 재설정합니다. 이 함수는 성공 시 `0`을 반환하고, 에러 시 `-1`을 반환합니다.

9.6.16 DateTime 객체

다양한 날짜와 시간 객체가 `datetime` 모듈에서 제공됩니다. 이 함수를 사용하기 전에, 헤더 파일 `datetime.h`가 소스에 포함되어야 하고 (`Python.h`가 포함하지 않음에 유의하십시오), 일반적으로 모듈 초기화 함수의 일부로 `PyDateTime_IMPORT` 매크로를 호출해야 합니다. 매크로는 C 구조체에 대한 포인터를 다음 매크로에서 사용되는 static 변수 `PyDateTimeAPI`에 넣습니다.

type **PyDateTime_Date**

This subtype of *PyObject* represents a Python date object.

type **PyDateTime_DateTime**

This subtype of *PyObject* represents a Python datetime object.

type **PyDateTime_Time**

This subtype of *PyObject* represents a Python time object.

type **PyDateTime_Delta**

This subtype of *PyObject* represents the difference between two datetime values.

PyTypeObject **PyDateTime_DateType**

This instance of *PyTypeObject* represents the Python date type; it is the same object as `datetime.date` in the Python layer.

PyTypeObject **PyDateTime_DateTimeType**

This instance of *PyTypeObject* represents the Python datetime type; it is the same object as `datetime.datetime` in the Python layer.

PyTypeObject **PyDateTime_TimeType**

This instance of *PyTypeObject* represents the Python time type; it is the same object as `datetime.time` in the Python layer.

PyTypeObject **PyDateTime_DeltaType**

This instance of *PyTypeObject* represents Python type for the difference between two datetime values; it is the same object as `datetime.timedelta` in the Python layer.

PyTypeObject **PyDateTime_TZInfoType**

This instance of *PyTypeObject* represents the Python time zone info type; it is the same object as `datetime.tzinfo` in the Python layer.

UTC 싱글톤에 액세스하기 위한 매크로:

PyObject ***PyDateTime_TimeZone_UTC**

UTC를 나타내는 시간대 싱글톤을 반환합니다, `datetime.timezone.utc`와 같은 객체입니다.

Added in version 3.7.

형 검사 매크로:

int **PyDate_Check**(*PyObject* *ob)

*ob*가 *PyDateTime_DateType* 형이거나 `PyDateTime_DateType`의 서브 형이면 참을 돌려줍니다. *ob*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyDate_CheckExact**(*PyObject* *ob)

*ob*가 *PyDateTime_DateType* 형이면 참을 돌려줍니다. *ob*는 `NULL`이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyDateTime_Check**(*PyObject* *ob)

*ob*가 *PyDateTime_DateTimeType* 형이거나 PyDateTime_DateTimeType의 서브 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyDateTime_CheckExact**(*PyObject* *ob)

*ob*가 *PyDateTime_DateTimeType* 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyTime_Check**(*PyObject* *ob)

*ob*가 *PyDateTime_TimeType* 형이거나 PyDateTime_TimeType의 서브 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyTime_CheckExact**(*PyObject* *ob)

*ob*가 *PyDateTime_TimeType* 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyDelta_Check**(*PyObject* *ob)

*ob*가 *PyDateTime_DeltaType* 형이거나 PyDateTime_DeltaType의 서브 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyDelta_CheckExact**(*PyObject* *ob)

*ob*가 *PyDateTime_DeltaType* 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyTZInfo_Check**(*PyObject* *ob)

*ob*가 *PyDateTime_TZInfoType* 형이거나 PyDateTime_TZInfoType의 서브 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int **PyTZInfo_CheckExact**(*PyObject* *ob)

*ob*가 *PyDateTime_TZInfoType* 형이면 참을 돌려줍니다. *ob*는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

객체를 만드는 매크로:

PyObject ***PyDate_FromDate**(int year, int month, int day)

반환값: 새 참조. 지정된 년, 월, 일의 datetime.date 객체를 반환합니다.

PyObject ***PyDateTime_FromDateAndTime**(int year, int month, int day, int hour, int minute, int second, int usecond)

반환값: 새 참조. 지정된 년, 월, 일, 시, 분, 초 및 마이크로초의 datetime.datetime 객체를 반환합니다.

PyObject ***PyDateTime_FromDateAndTimeAndFold**(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

반환값: 새 참조. 지정된 년, 월, 일, 시, 분, 초, 마이크로초 및 fold의 datetime.datetime 객체를 반환합니다.

Added in version 3.6.

PyObject ***PyTime_FromTime**(int hour, int minute, int second, int usecond)

반환값: 새 참조. 지정된 시, 분, 초 및 마이크로초의 datetime.time 객체를 반환합니다.

PyObject ***PyTime_FromTimeAndFold**(int hour, int minute, int second, int usecond, int fold)

반환값: 새 참조. 지정된 시, 분, 초, 마이크로초 및 fold의 datetime.time 객체를 반환합니다.

Added in version 3.6.

PyObject ***PyDelta_FromDSU**(int days, int seconds, int useconds)

반환값: 새 참조. 지정된 일, 초 및 마이크로초 수를 나타내는 datetime.timedelta 객체를 반환합니다. 결과 마이크로초와 초가 datetime.timedelta 객체에 관해 설명된 범위에 있도록 정규화가 수행됩니다.

PyObject ***PyTimeZone_FromOffset**(*PyObject* *offset)

반환값: 새 참조. *offset* 인자로 나타내지는 이름이 없는 고정 오프셋의 `datetime.timezone` 객체를 돌려줍니다.

Added in version 3.7.

PyObject ***PyTimeZone_FromOffsetAndName**(*PyObject* *offset, *PyObject* *name)

반환값: 새 참조. *offset* 인자와 tzname *name*으로 나타내지는 고정 오프셋의 `datetime.timezone` 객체를 돌려줍니다.

Added in version 3.7.

날짜 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스(가령 *PyDateTime_DateTime*)를 포함하여 *PyDateTime_Date*의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며, 형은 검사하지 않습니다:

int **PyDateTime_GET_YEAR**(*PyDateTime_Date* *o)

양의 int로, 년을 반환합니다.

int **PyDateTime_GET_MONTH**(*PyDateTime_Date* *o)

1에서 12까지의 int로, 월을 반환합니다.

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

1에서 31까지의 int로, 일을 반환합니다.

날짜 시간 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스를 포함하여 *PyDateTime_DateTime*의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며, 형은 검사하지 않습니다.:

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)

0부터 23까지의 int로, 시를 반환합니다.

int **PyDateTime_DATE_GET_MINUTE**(*PyDateTime_DateTime* *o)

0부터 59까지의 int로, 분을 반환합니다.

int **PyDateTime_DATE_GET_SECOND**(*PyDateTime_DateTime* *o)

0부터 59까지의 int로, 초를 반환합니다.

int **PyDateTime_DATE_GET_MICROSECOND**(*PyDateTime_DateTime* *o)

0부터 999999까지의 int로, 마이크로초를 반환합니다.

int **PyDateTime_DATE_GET_FOLD**(*PyDateTime_DateTime* *o)

0에서 1까지의 int로, 폴드를 반환합니다.

Added in version 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)

Return the tzinfo (which may be `None`).

Added in version 3.10.

시간 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스를 포함하여 *PyDateTime_Time*의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며 형은 검사하지 않습니다:

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)

0부터 23까지의 int로, 시를 반환합니다.

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)

0부터 59까지의 int로, 분을 반환합니다.

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)

0부터 59까지의 int로, 초를 반환합니다.

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)

0부터 999999까지의 int로, 마이크로초를 반환합니다.

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)

0에서 1까지의 int로, 폴드를 반환합니다.

Added in version 3.6.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)

Return the tzinfo (which may be `None`).

Added in version 3.10.

시간 델타 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스를 포함하여 *PyDateTime_Delta*의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며 형은 검사하지 않습니다.:

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)

-999999999에서 999999999까지의 int로, 일 수를 반환합니다.

Added in version 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)

0부터 86399까지의 int로, 초 수를 반환합니다.

Added in version 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)

0에서 999999까지의 int로, 마이크로초 수를 반환합니다.

Added in version 3.3.

DB API를 구현하는 모듈의 편의를 위한 매크로:

PyObject ***PyDateTime_FromTimestamp**(*PyObject* *args)

반환값: 새 참조. `datetime.datetime.fromtimestamp()`에 전달하는 데 적합한 인자 튜플로 새 `datetime.datetime` 객체를 만들고 반환합니다.

PyObject ***PyDate_FromTimestamp**(*PyObject* *args)

반환값: 새 참조. `datetime.date.fromtimestamp()`에 전달하는 데 적합한 인자 튜플로 새 `datetime.date` 객체를 만들고 반환합니다.

9.6.17 Objects for Type Hinting

Various built-in types for type hinting are provided. Currently, two types exist – GenericAlias and Union. Only `GenericAlias` is exposed to C.

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

Part of the 안정 ABI 버전 *3.9* 이후로. Create a GenericAlias object. Equivalent to calling the Python class `types.GenericAlias`. The *origin* and *args* arguments set the `GenericAlias` 's `__origin__` and `__args__` attributes respectively. *origin* should be a *PyTypeObject**, and *args* can be a *PyTupleObject** or any `PyObject*`. If *args* passed is not a tuple, a 1-tuple is automatically constructed and `__args__` is set to `(args,)`. Minimal checking is done for the arguments, so the function will succeed even if *origin* is not a type. The `GenericAlias` 's `__parameters__` attribute is constructed lazily from `__args__`. On failure, an exception is raised and `NULL` is returned.

Here' s an example of how to make an extension type generic:

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

> **더 보기**
>
> The data model method `__class_getitem__()`.

Added in version 3.9.

PyTypeObject **Py_GenericAliasType**

Part of the 안정 ABI 버전 *3.9* 이후로. The C type of the object returned by *Py_GenericAlias()*. Equivalent to `types.GenericAlias` in Python.

Added in version 3.9.

CHAPTER 10

초기화, 파이널리제이션 및 스레드

See *Python Initialization Configuration* for details on how to configure the interpreter prior to initialization.

10.1 파이썬 초기화 전

파이썬을 내장한 응용 프로그램에서는, 다른 파이썬/C API 함수를 사용하기 전에 *Py_Initialize()* 함수를 호출해야 합니다; 몇 가지 함수와 전역 구성 변수는 예외입니다.

파이썬이 초기화되기 전에 다음 함수를 안전하게 호출할 수 있습니다:

- Functions that initialize the interpreter:
  - *Py_Initialize()*
  - *Py_InitializeEx()*
  - *Py_InitializeFromConfig()*
  - *Py_BytesMain()*
  - *Py_Main()*
  - the runtime pre-initialization functions covered in 파이썬 초기화 구성
- 구성 함수:
  - *PyImport_AppendInittab()*
  - *PyImport_ExtendInittab()*
  - PyInitFrozenExtensions()
  - *PyMem_SetAllocator()*
  - *PyMem_SetupDebugHooks()*
  - *PyObject_SetArenaAllocator()*
  - *Py_SetProgramName()*
  - *Py_SetPythonHome()*
  - *PySys_ResetWarnOptions()*
  - the configuration functions covered in 파이썬 초기화 구성

- 정보 함수:
  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
  - *Py_IsInitialized()*
- 유틸리티:
  - *Py_DecodeLocale()*
  - the status reporting and utility functions covered in 파이썬 초기화 구성
- 메모리 할당자:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*
- Synchronization:
  - *PyMutex_Lock()*
  - *PyMutex_Unlock()*

참고

Despite their apparent similarity to some of the functions listed above, the following functions **should not be called** before the interpreter has been initialized: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()*, *PyEval_InitThreads()*, and *Py_RunMain()*.

10.2 전역 구성 변수

파이썬에는 다양한 기능과 옵션을 제어하기 위한 전역 구성 변수가 있습니다. 기본적으로, 이러한 플래그는 명령 줄 옵션에 의해 제어됩니다.

옵션에 의해 플래그가 설정되면, 플래그 값은 옵션이 설정된 횟수입니다. 예를 들어, `-b`는 *Py_BytesWarningFlag*를 1로 설정하고 `-bb`는 *Py_BytesWarningFlag*를 2로 설정합니다.

int **Py_BytesWarningFlag**

This API is kept for backward compatibility: setting *PyConfig.bytes_warning* should be used instead, see *Python Initialization Configuration*.

`bytes`나 `bytearray`와 `str`을, 또는 `bytes`를 `int`와 비교할 때 경고를 발행합니다. `2`보다 크거나 같으면 에러를 발행합니다.

`-b` 옵션으로 설정합니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_DebugFlag**

This API is kept for backward compatibility: setting *PyConfig.parser_debug* should be used instead, see *Python Initialization Configuration*.

구문 분석기 디버깅 출력을 켭니다 (전문가 전용, 컴파일 옵션에 의존합니다).

`-d` 옵션과 `PYTHONDEBUG` 환경 변수로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_DontWriteBytecodeFlag**

This API is kept for backward compatibility: setting *PyConfig.write_bytecode* should be used instead, see *Python Initialization Configuration*.

0이 아닌 값으로 설정하면, 파이썬은 소스 모듈을 임포트 할 때 `.pyc` 파일을 쓰려고 하지 않습니다.

`-B` 옵션과 `PYTHONDONTWRITEBYTECODE` 환경 변수로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_FrozenFlag**

This API is kept for backward compatibility: setting *PyConfig.pathconfig_warnings* should be used instead, see *Python Initialization Configuration*.

*Py_GetPath()*에서 모듈 검색 경로를 계산할 때 에러 메시지를 표시하지 않습니다.

Private flag used by `_freeze_module` and `frozenmain` programs.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_HashRandomizationFlag**

This API is kept for backward compatibility: setting *PyConfig.hash_seed* and *PyConfig.use_hash_seed* should be used instead, see *Python Initialization Configuration*.

`PYTHONHASHSEED` 환경 변수가 비어 있지 않은 문자열로 설정되면 `1`로 설정합니다.

플래그가 0이 아니면, `PYTHONHASHSEED` 환경 변수를 읽어 비밀 해시 시드를 초기화합니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_IgnoreEnvironmentFlag**

This API is kept for backward compatibility: setting *PyConfig.use_environment* should be used instead, see *Python Initialization Configuration*.

Ignore all `PYTHON*` environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

`-E`와 `-I` 옵션으로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_InspectFlag**

This API is kept for backward compatibility: setting *PyConfig.inspect* should be used instead, see *Python Initialization Configuration*.

스크립트가 첫 번째 인자로 전달되거나 `-c` 옵션을 사용할 때, `sys.stdin`가 터미널로 보이지 않더라도 스크립트나 명령을 실행한 후 대화 형 모드로 들어갑니다.

`-i` 옵션과 `PYTHONINSPECT` 환경 변수로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_InteractiveFlag**

This API is kept for backward compatibility: setting *PyConfig.interactive* should be used instead, see *Python Initialization Configuration*.

`-i` 옵션으로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_IsolatedFlag**

This API is kept for backward compatibility: setting *PyConfig.isolated* should be used instead, see *Python Initialization Configuration*.

격리 모드로 파이썬을 실행합니다. 격리 모드에서 sys.path는 스크립트의 디렉터리도 사용자의 site-packages 디렉터리도 포함하지 않습니다.

-I 옵션으로 설정됩니다.

Added in version 3.4.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_LegacyWindowsFSEncodingFlag**

This API is kept for backward compatibility: setting *PyPreConfig.legacy_windows_fs_encoding* should be used instead, see *Python Initialization Configuration*.

If the flag is non-zero, use the mbcs encoding with replace error handler, instead of the UTF-8 encoding with surrogatepass error handler, for the *filesystem encoding and error handler*.

PYTHONLEGACYWINDOWSFSENCODING 환경 변수가 비어 있지 않은 문자열로 설정되면 1로 설정합니다.

자세한 내용은 **PEP 529**를 참조하십시오.

가용성: Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_LegacyWindowsStdioFlag**

This API is kept for backward compatibility: setting *PyConfig.legacy_windows_stdio* should be used instead, see *Python Initialization Configuration*.

If the flag is non-zero, use io.FileIO instead of io._WindowsConsoleIO for sys standard streams.

PYTHONLEGACYWINDOWSSTDIO 환경 변수가 비어 있지 않은 문자열로 설정되면 1로 설정합니다.

자세한 내용은 **PEP 528**을 참조하십시오.

가용성: Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_NoSiteFlag**

This API is kept for backward compatibility: setting *PyConfig.site_import* should be used instead, see *Python Initialization Configuration*.

모듈 site 임포트와 이에 수반되는 sys.path의 사이트 종속적인 조작을 비활성화합니다. 또한 나중에 site를 명시적으로 임포트 할 때도 이러한 조작을 비활성화합니다 (트리거 하려면 site.main() 을 호출하십시오).

-S 옵션으로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_NoUserSiteDirectory**

This API is kept for backward compatibility: setting *PyConfig.user_site_directory* should be used instead, see *Python Initialization Configuration*.

사용자 site-packages 디렉터리를 sys.path에 추가하지 않습니다.

-s와 -I 옵션, 그리고 PYTHONNOUSERSITE 환경 변수로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_OptimizeFlag**

This API is kept for backward compatibility: setting *PyConfig.optimization_level* should be used instead, see *Python Initialization Configuration*.

-O 옵션과 PYTHONOPTIMIZE 환경 변수로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_QuietFlag**

This API is kept for backward compatibility: setting *PyConfig.quiet* should be used instead, see *Python Initialization Configuration*.

대화형 모드에서도 저작권과 버전 메시지를 표시하지 않습니다.

-q 옵션으로 설정됩니다.

Added in version 3.2.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_UnbufferedStdioFlag**

This API is kept for backward compatibility: setting *PyConfig.buffered_stdio* should be used instead, see *Python Initialization Configuration*.

stdout과 stderr 스트림을 버퍼링 해제하도록 강제합니다.

-u 옵션과 PYTHONUNBUFFERED 환경 변수로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_VerboseFlag**

This API is kept for backward compatibility: setting *PyConfig.verbose* should be used instead, see *Python Initialization Configuration*.

모듈이 초기화될 때마다, 로드된 위치(파일명이나 내장 모듈)를 표시하는 메시지를 인쇄합니다. 2 보다 크거나 같으면, 모듈을 검색할 때 검사되는 각 파일에 대한 메시지를 인쇄합니다. 또한 종료 시 모듈 정리에 대한 정보를 제공합니다.

-v 옵션과 PYTHONVERBOSE 환경 변수로 설정됩니다.

Deprecated since version 3.12, will be removed in version 3.15.

10.3 인터프리터 초기화와 파이널리제이션

void **Py_Initialize**()

Part of the 안정 ABI. 파이썬 인터프리터를 초기화합니다. 파이썬을 내장하는 응용 프로그램에서는, 다른 파이썬/C API 함수를 사용하기 전에 호출해야 합니다; 몇 가지 예외는 파이썬 초기화 전을 참조하십시오.

This initializes the table of loaded modules (sys.modules), and creates the fundamental modules builtins, __main__ and sys. It also initializes the module search path (sys.path). It does not set sys.argv; use the *Python Initialization Configuration* API for that. This is a no-op when called for a second time (without calling *Py_FinalizeEx()* first). There is no return value; it is a fatal error if the initialization fails.

Use *Py_InitializeFromConfig()* to customize the *Python Initialization Configuration*.

> **참고**
>
> 윈도우에서, 콘솔 모드를 O_TEXT에서 O_BINARY로 변경합니다, C 런타임을 사용하는 콘솔의 비 파이썬 사용에도 영향을 미칩니다.

void **Py_InitializeEx**(int initsigs)

Part of the 안정 ABI. This function works like *Py_Initialize()* if *initsigs* is 1. If *initsigs* is 0, it skips initialization registration of signal handlers, which may be useful when CPython is embedded as part of a larger application.

Use *Py_InitializeFromConfig()* to customize the *Python Initialization Configuration*.

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

Initialize Python from *config* configuration, as described in *PyConfig*를 사용한 초기화.

See the 파이썬 초기화 구성 section for details on pre-initializing the interpreter, populating the runtime configuration structure, and querying the returned status structure.

int **Py_IsInitialized**()

Part of the 안정 ABI. 파이썬 인터프리터가 초기화되었으면 참(0이 아님)을 반환하고, 그렇지 않으면 거짓(0)을 반환합니다. `Py_FinalizeEx()` 가 호출된 후, `Py_Initialize()`가 다시 호출될 때까지 거짓을 반환합니다.

int **Py_IsFinalizing**()

Part of the 안정 ABI 버전 *3.13* 이후로. Return true (non-zero) if the main Python interpreter is *shutting down*. Return false (zero) otherwise.

Added in version 3.13.

int **Py_FinalizeEx**()

Part of the 안정 ABI 버전 *3.6* 이후로. Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. This is a no-op when called for a second time (without calling `Py_Initialize()` again first).

Since this is the reverse of `Py_Initialize()`, it should be called in the same thread with the same interpreter active. That means the main thread and the main interpreter. This should never be called while `Py_RunMain()` is running.

Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

Note that Python will do a best effort at freeing all memory allocated by the Python interpreter. Therefore, any C-Extension should make sure to correctly clean up all of the preveiously allocated PyObjects before using them in subsequent calls to `Py_Initialize()`. Otherwise it could introduce vulnerabilities and incorrect behavior.

이 함수는 여러 가지 이유로 제공됩니다. 내장 응용 프로그램이 응용 프로그램 자체를 다시 시작하지 않고 파이썬을 다시 시작하고 싶을 수 있습니다. 동적으로 로드할 수 있는 라이브러리(또는 DLL)에서 파이썬 인터프리터를 로드한 응용 프로그램은 DLL을 언로드 하기 전에 파이썬이 할당한 모든 메모리를 해제하고 싶을 수 있습니다. 응용 프로그램에서 메모리 누수를 찾는 동안 개발자는 응용 프로그램을 종료하기 전에 파이썬에서 할당한 모든 메모리를 해제하고 싶을 것입니다.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Interned strings will all be deallocated regardless of their reference count. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls `Py_Initialize()` and `Py_FinalizeEx()` more than once. `Py_FinalizeEx()` must not be called recursively from within itself. Therefore, it must not be called by any code that may be run as part of the interpreter shutdown process, such as `atexit` handlers, object finalizers, or any code that may be run while flushing the stdout and stderr files.

인자 없이 감사 이벤트 `cpython._PySys_ClearAuditHooks` 를 발생시킵니다.

Added in version 3.6.

void **Py_Finalize**()

Part of the 안정 ABI. 이것은 `Py_FinalizeEx()`의 이전 버전과 호환되는 반환 값을 무시하는 버전입니다.

int **Py_BytesMain**(int argc, char **argv)

Part of the 안정 ABI 버전 *3.8* 이후로. Similar to `Py_Main()` but *argv* is an array of bytes strings, allowing the calling application to delegate the text decoding step to the CPython runtime.

Added in version 3.8.

int **Py_Main**(int argc, wchar_t **argv)

Part of the 안정 ABI. The main program for the standard interpreter, encapsulating a full initialization/finalization cycle, as well as additional behaviour to implement reading configurations settings from the environment and command line, and then executing `__main__` in accordance with using-on-cmdline.

This is made available for programs which wish to support the full CPython command line interface, rather than just embedding a Python runtime in a larger application.

The *argc* and *argv* parameters are similar to those which are passed to a C program's `main()` function, except that the *argv* entries are first converted to `wchar_t` using *Py_DecodeLocale()*. It is also important to note that the argument list entries may be modified to point to strings other than those passed in (however, the contents of the strings pointed to by the argument list are not modified).

The return value is `2` if the argument list does not represent a valid Python command line, and otherwise the same as *Py_RunMain()*.

In terms of the CPython runtime configuration APIs documented in the *runtime configuration* section (and without accounting for error handling), `Py_Main` is approximately equivalent to:

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

In normal usage, an embedding application will call this function *instead* of calling *Py_Initialize()*, *Py_InitializeEx()* or *Py_InitializeFromConfig()* directly, and all settings will be applied as described elsewhere in this documentation. If this function is instead called *after* a preceding runtime initialization API call, then exactly which environmental and command line configuration settings will be updated is version dependent (as it depends on which settings correctly support being modified after they have already been set once when the runtime was first initialized).

int **Py_RunMain**(void)

Executes the main module in a fully configured CPython runtime.

Executes the command (*PyConfig.run_command*), the script (*PyConfig.run_filename*) or the module (*PyConfig.run_module*) specified on the command line or in the configuration. If none of these values are set, runs the interactive Python prompt (REPL) using the `__main__` module's global namespace.

If *PyConfig.inspect* is not set (the default), the return value will be `0` if the interpreter exits normally (that is, without raising an exception), the exit status of an unhandled `SystemExit`, or `1` for any other unhandled exception.

If *PyConfig.inspect* is set (such as when the `-i` option is used), rather than returning when the interpreter exits, execution will instead resume in an interactive Python prompt (REPL) using the `__main__` module's global namespace. If the interpreter exited with an exception, it is immediately raised in the REPL session. The function return value is then determined by the way the *REPL session* terminates: `0`, `1`, or the status of a `SystemExit`, as specified above.

This function always finalizes the Python interpreter before it returns.

See *Python Configuration* for an example of a customized Python that always runs in isolated mode using *Py_RunMain()*.

int **PyUnstable_AtExit**(*PyInterpreterState* *interp, void (*func)(void*), void *data)

이것은 불안정 *API*. It may change without warning in minor releases.

Register an `atexit` callback for the target interpreter *interp*. This is similar to *Py_AtExit()*, but takes an explicit interpreter and data pointer for the callback.

There must be an *attached thread state* for *interp*.

Added in version 3.13.

10.4 프로세스 전체 매개 변수

void **Py_SetProgramName**(const wchar_t *name)

Part of the 안정 ABI. This API is kept for backward compatibility: setting *PyConfig.program_name* should be used instead, see *Python Initialization Configuration*.

(호출된다면) 이 함수는 *Py_Initialize()*가 처음으로 호출되기 전에 호출되어야 합니다. 인터프리터에게 프로그램의 `main()` 함수에 대한 `argv[0]` 인자의 값을 알려줍니다 (와이드 문자로 변환됩니다). 이것은 *Py_GetPath()*와 아래의 다른 함수에서 인터프리터 실행 파일과 관련된 파이썬 런타임 라이브러리를 찾는 데 사용됩니다. 기본값은 `'python'`입니다. 인자는 프로그램을 실행하는 동안 내용이 변경되지 않는 정적 저장소의 0으로 끝나는 와이드 문자열을 가리켜야 합니다. 파이썬 인터프리터의 코드는 이 저장소의 내용을 변경하지 않습니다.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t ***Py_GetProgramName**()

Part of the 안정 ABI. Return the program name set with *PyConfig.program_name*, or the default. The returned string points into static storage; the caller should not modify its value.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

버전 3.10에서 변경: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("executable")* (`sys.executable`) instead.

wchar_t ***Py_GetPrefix**()

Part of the 안정 ABI. Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with *PyConfig.program_name* and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level `Makefile` and the `--prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.base_prefix`. It is only useful on Unix. See also the next function.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

버전 3.10에서 변경: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("base_prefix")* (`sys.base_prefix`) instead. Use *PyConfig_Get("prefix")* (`sys.prefix`) if virtual environments need to be handled.

wchar_t ***Py_GetExecPrefix**()

Part of the 안정 ABI. Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with *PyConfig.program_name* and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the exec-prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level `Makefile` and the `--exec-prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.base_exec_prefix`. It is only useful on Unix.

배경: exec-prefix는 플랫폼 종속적 파일(가령 실행 파일과 공유 라이브러리)이 다른 디렉터리 트리에 설치될 때 prefix와 다릅니다. 일반 설치에서, 플랫폼 종속적 파일은 `/usr/local/plat` 서브 트리에 설치되고 플랫폼 독립적 파일은 `/usr/local`에 설치될 수 있습니다.

일반적으로 말해서, 플랫폼은 하드웨어와 소프트웨어 제품군의 조합입니다, 예를 들어 Solaris 2.x 운영 체제를 실행하는 Sparc 기계들은 같은 플랫폼으로 간주하지만, Solaris 2.x를 실행하는 Intel 기계는 다른 플랫폼이며, 리눅스를 실행하는 Intel 기계는 또 다른 플랫폼입니다. 같은 운영 체제의 서로 다른 주 개정판도 일반적으로 다른 플랫폼을 형성합니다. 비 유닉스 운영 체제는 다른 이야기입니다; 이러한 시스템의 설치 전략이 너무 다르기 때문에 prefix와 exec-prefix는 의미가 없으며, 빈 문자열로 설정됩니다. 컴파일된 파이썬 바이트 코드 파일은 플랫폼 독립적임에 유의하십시오 (그러나 이들을 컴파일하는데 사용된 파이썬 버전에는 종속적입니다!).

시스템 관리자는 `/usr/local/plat`을 각 플랫폼에 대해 다른 파일 시스템으로 사용하면서 플랫폼 간에 `/usr/local`을 공유하도록 **mount** 나 **automount** 프로그램을 구성하는 방법을 알 것입니다.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

버전 3.10에서 변경: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("base_exec_prefix")* (`sys.base_exec_prefix`) instead. Use *PyConfig_Get("exec_prefix")* (`sys.exec_prefix`) if virtual environments need to be handled.

wchar_t ***Py_GetProgramFullPath**()

Part of the 안정 ABI. Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by *PyConfig.program_name*). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

버전 3.10에서 변경: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("executable")* (`sys.executable`) instead.

wchar_t ***Py_GetPath**()

Part of the 안정 ABI. Return the default module search path; this is computed from the program name (set by *PyConfig.program_name*) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and macOS, `';'` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

버전 3.10에서 변경: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("module_search_paths")* (`sys.path`) instead.

const char ***Py_GetVersion**()

Part of the 안정 ABI. 이 파이썬 인터프리터의 버전을 반환합니다. 이것은 다음과 같은 문자열입니다

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

See also the *Py_Version* constant.

const char ***Py_GetPlatform**()

Part of the 안정 ABI. Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On macOS, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char *Py_GetCopyright()

Part of the 안정 ABI. 현재 파이썬 버전에 대한 공식 저작권 문자열을 반환합니다, 예를 들어

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 sys.copyright로 사용할 수 있습니다.

const char *Py_GetCompiler()

Part of the 안정 ABI. 현재 파이썬 버전을 빌드하는 데 사용된 컴파일러 표시를 대괄호 감싸서 반환합니다, 예를 들면:

```
"[GCC 2.7.2.2]"
```

반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 변수 sys.version의 일부로 제공됩니다.

const char *Py_GetBuildInfo()

Part of the 안정 ABI. 현재 파이썬 인터프리터 인스턴스의 시퀀스 번호와 빌드 날짜 및 시간에 대한 정보를 반환합니다, 예를 들어

```
"#67, Aug  1 1997, 22:34:28"
```

반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 변수 sys.version의 일부로 제공됩니다.

void PySys_SetArgvEx(int argc, wchar_t **argv, int updatepath)

Part of the 안정 ABI. This API is kept for backward compatibility: setting *PyConfig.argv*, *PyConfig.parse_argv* and *PyConfig.safe_path* should be used instead, see *Python Initialization Configuration*.

argc 및 *argv*에 기반해서 sys.argv를 설정합니다. 이 매개 변수는 프로그램의 main() 함수에 전달된 것과 유사하지만, 첫 번째 항목이 파이썬 인터프리터를 호스팅하는 실행 파일이 아니라 실행될 스크립트 파일을 참조해야 한다는 차이점이 있습니다. 실행할 스크립트가 없으면, *argv*의 첫 번째 항목은 빈 문자열일 수 있습니다. 이 함수가 sys.argv 초기화에 실패하면, *Py_FatalError()*를 사용하여 치명적인 조건을 표시합니다.

*updatepath*가 0이면, 여기까지가 이 함수가 하는 모든 일입니다. *updatepath*가 0이 아니면, 함수는 다음 알고리즘에 따라 sys.path도 수정합니다:

- 기존 스크립트의 이름이 argv[0]으로 전달되면, 스크립트가 있는 디렉터리의 절대 경로가 sys.path 앞에 추가됩니다.
- 그렇지 않으면 (즉, *argc*가 0이거나 argv[0]이 기존 파일 이름을 가리키지 않으면), sys.path 앞에 빈 문자열이 추가됩니다, 이는 현재 작업 디렉터리(".")를 앞에 추가하는 것과 같습니다.

Use *Py_DecodeLocale()* to decode a bytes string to get a wchar_t* string.

See also *PyConfig.orig_argv* and *PyConfig.argv* members of the *Python Initialization Configuration*.

> **참고**
>
> It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass 0 as *updatepath*, and update sys.path themselves if desired. See **CVE 2008-5983**.
>
> 3.1.3 이전 버전에서는, *PySys_SetArgv()*를 호출한 후 첫 번째 sys.path 요소를 수동으로 제거하여 같은 효과를 얻을 수 있습니다, 예를 들어 다음을 사용하여:
>
> ```
> PyRun_SimpleString("import sys; sys.path.pop(0)\n");
> ```

Added in version 3.1.3.

Deprecated since version 3.11, will be removed in version 3.15.

void **PySys_SetArgv**(int argc, wchar_t **argv)

Part of the 안정 ABI. This API is kept for backward compatibility: setting *PyConfig.argv* and *PyConfig.parse_argv* should be used instead, see *Python Initialization Configuration*.

이 함수는 **python** 인터프리터가 `-I`로 시작되지 않는 한 *updatepath*가 `1`로 설정된 *PySys_SetArgvEx()*처럼 작동합니다.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

See also *PyConfig.orig_argv* and *PyConfig.argv* members of the *Python Initialization Configuration*.

버전 3.4에서 변경: *updatepath* 값은 `-I`에 따라 다릅니다.

Deprecated since version 3.11, will be removed in version 3.15.

void **Py_SetPythonHome**(const wchar_t *home)

Part of the 안정 ABI. This API is kept for backward compatibility: setting *PyConfig.home* should be used instead, see *Python Initialization Configuration*.

기본 "홈" 디렉터리, 즉 표준 파이썬 라이브러리의 위치를 설정합니다. 인자 문자열의 의미는 `PYTHONHOME`을 참조하십시오.

인자는 프로그램을 실행하는 동안 내용이 변경되지 않는 정적 저장소에 있는 0으로 끝나는 문자열을 가리켜야 합니다. 파이썬 인터프리터의 코드는 이 저장소의 내용을 변경하지 않습니다.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t ***Py_GetPythonHome**()

Part of the 안정 ABI. Return the default "home", that is, the value set by *PyConfig.home*, or the value of the `PYTHONHOME` environment variable if it is set.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

버전 3.10에서 변경: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("home")* or the `PYTHONHOME` environment variable instead.

10.5 스레드 상태와 전역 인터프리터 록

Unless on a *free-threaded* build of *CPython*, the Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

따라서, *GIL*을 획득한 스레드만 파이썬 객체에서 작동하거나 파이썬/C API 함수를 호출할 수 있다는 규칙이 있습니다. 동시 실행을 모방하기 위해 인터프리터는 정기적으로 스레드 전환을 시도합니다(`sys.setswitchinterval()`을 참조하십시오). 록은 파일 읽기나 쓰기와 같은 잠재적인 블로킹 I/O 연산에 대해서도 해제되므로, 그동안 다른 파이썬 스레드가 실행될 수 있습니다.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*, known as a *thread state*. Each OS thread has a thread-local pointer to a *PyThreadState*; a thread state referenced by this pointer is considered to be *attached*.

A thread can only have one *attached thread state* at a time. An attached thread state is typically analogous with holding the *GIL*, except on *free-threaded* builds. On builds with the *GIL* enabled, *attaching* a thread state will block until the *GIL* can be acquired. However, even on builds with the *GIL* disabled, it is still required to have an attached thread state to call most of the C API.

In general, there will always be an *attached thread state* when using Python's C API. Only in some specific cases (such as in a *Py_BEGIN_ALLOW_THREADS* block) will the thread not have an attached thread state. If uncertain, check if *PyThreadState_GetUnchecked()* returns `NULL`.

10.5.1 Detaching the thread state from extension code

Most extension code manipulating the *thread state* has the following simple structure:

```
Save the thread state in a local variable.
... Do some blocking I/O operation ...
Restore the thread state from the local variable.
```

이것은 매우 일반적이어서 이를 단순화하기 위해 한 쌍의 매크로가 존재합니다:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

Py_BEGIN_ALLOW_THREADS 매크로는 새 블록을 열고 숨겨진 지역 변수를 선언합니다; *Py_END_ALLOW_THREADS* 매크로는 블록을 닫습니다.

위의 블록은 다음 코드로 확장됩니다:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work:

The *attached thread state* holds the *GIL* for the entire interpreter. When detaching the *attached thread state*, the *GIL* is released, allowing other threads to attach a thread state to their own thread, thus getting the *GIL* and can start executing. The pointer to the prior *attached thread state* is stored as a local variable. Upon reaching *Py_END_ALLOW_THREADS*, the thread state that was previously *attached* is passed to *PyEval_RestoreThread()*. This function will block until another releases its *thread state*, thus allowing the old *thread state* to get re-attached and the C API can be called again.

For *free-threaded* builds, the *GIL* is normally out of the question, but detaching the *thread state* is still required for blocking I/O and long operations. The difference is that threads don' t have to wait for the *GIL* to be released to attach their thread state, allowing true multi-core parallelism.

참고

Calling system I/O functions is the most common use case for detaching the *thread state*, but it can also be useful before calling long-running computations which don' t need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules detach the *thread state* when compressing or hashing data.

10.5.2 파이썬이 만들지 않은 스레드

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don' t hold the *GIL*, because they don' t have an *attached thread state*.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating an *attached thread state* before you can start using the Python/C API. When you are done, you should detach the *thread state*, and finally free it.

*PyGILState_Ensure()*와 *PyGILState_Release()* 함수는 위의 모든 작업을 자동으로 수행합니다. C 스레드에서 파이썬을 호출하는 일반적인 관용구는 다음과 같습니다:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by *`Py_Initialize()`*). Python supports the creation of additional interpreters (using *`Py_NewInterpreter()`*), but mixing multiple interpreters and the `PyGILState_*` API is unsupported. This is because *`PyGILState_Ensure()`* and similar functions default to *attaching* a *thread state* for the main interpreter, meaning that the thread can' t safely interact with the calling subinterpreter.

10.5.3 Supporting subinterpreters in non-Python threads

If you would like to support subinterpreters with non-Python created threads, you must use the `PyThreadState_*` API instead of the traditional `PyGILState_*` API.

In particular, you must store the interpreter state from the calling function and pass it to *`PyThreadState_New()`*, which will ensure that the *thread state* is targeting the correct interpreter:

```
/* The return value of PyInterpreterState_Get() from the
   function that created this thread. */
PyInterpreterState *interp = ThreadData->interp;
PyThreadState *tstate = PyThreadState_New(interp);
PyThreadState_Swap(tstate);

/* GIL of the subinterpreter is now held.
   Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Destroy the thread state. No Python API allowed beyond this point. */
PyThreadState_Clear(tstate);
PyThreadState_DeleteCurrent();
```

10.5.4 fork()에 대한 주의 사항

스레드에 대해 주목해야 할 또 다른 중요한 점은 C `fork()` 호출 시 스레드의 동작입니다. `fork()`를 사용하는 대부분의 시스템에서는, 프로세스가 포크한 후에 포크를 발행한 스레드만 존재합니다. 이는 록을 처리해야 하는 방법과 CPython 런타임에 저장된 모든 상태 모두에 구체적인 영향을 미칩니다.

The fact that only the "current" thread remains means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python' s internal locks being held by a thread that is defunct after the fork. *`PyOS_AfterFork_Child()`* tries to reset the necessary locks, but is not always able to.

다른 모든 스레드가 사라진다는 사실은 또한 CPython의 런타임 상태가 `os.fork()`와 마찬가지로 적절하게 정리되어야 함을 의미합니다. 이것은 현재 인터프리터와 다른 모든 *`PyInterpreterState`* 객체에 속하는 다른 모든 *`PyThreadState`* 객체를 파이널리제이션 하는 것을 의미합니다. 이것과 "메인" 인터프리터의 특수한 특성으로 인해, `fork()`는 CPython 전역 런타임이 원래 초기화된 인터프리터의 "메인" 스레드에서만 호출되어야 합니다. 유일한 예외는 `exec()`가 그 후에 즉시 호출되는 경우입니다.

10.5.5 Cautions regarding runtime finalization

In the late stage of *interpreter shutdown*, after attempting to wait for non-daemon threads to exit (though this can be interrupted by `KeyboardInterrupt`) and running the `atexit` functions, the runtime is marked as *finalizing*: *`Py_IsFinalizing()`* and `sys.is_finalizing()` return true. At this point, only the *finalization thread* that initiated finalization (typically the main thread) is allowed to acquire the *GIL*.

If any thread, other than the finalization thread, attempts to attach a *thread state* during finalization, either explicitly or implicitly, the thread enters **a permanently blocked state** where it remains until the program exits. In most cases this is harmless, but this can result in deadlock if a later stage of finalization attempts to acquire a lock owned by the blocked thread, or otherwise waits on the blocked thread.

Gross? Yes. This prevents random crashes and/or unexpectedly skipped C++ finalizations further up the call stack when such threads were forcibly exited here in CPython 3.13 and earlier. The CPython runtime *thread state* C APIs have never had any error reporting or handling expectations at *thread state* attachment time that would' ve allowed for graceful exit from this situation. Changing that would require new stable C APIs and rewriting the majority of C code in the CPython ecosystem to use those with error handling.

10.5.6 고수준 API

다음은 C 확장 코드를 작성하거나 파이썬 인터프리터를 내장할 때 가장 일반적으로 사용되는 형과 함수입니다:

type **PyInterpreterState**

Part of the 제한된 API (*as an opaque struct*). 이 자료 구조는 여러 협력 스레드가 공유하는 상태를 나타냅니다. 같은 인터프리터에 속하는 스레드는 모듈 관리와 몇 가지 다른 내부 항목을 공유합니다. 이 구조체에는 공개 멤버가 없습니다.

다른 인터프리터에 속한 스레드는 사용 가능한 메모리, 열린 파일 기술자 등과 같은 프로세스 상태를 제외하고는, 처음에는 아무것도 공유하지 않습니다. 전역 인터프리터 록은 어떤 인터프리터에 속해 있는지에 관계없이 모든 스레드에서 공유됩니다.

버전 3.12에서 변경: **PEP 684** introduced the possibility of a *per-interpreter GIL*. See *`Py_NewInterpreterFromConfig()`*.

type **PyThreadState**

Part of the 제한된 API (*as an opaque struct*). This data structure represents the state of a single thread. The only public data member is:

PyInterpreterState ***interp**

This thread' s interpreter state.

void **PyEval_InitThreads**()

Part of the 안정 ABI. 아무것도 하지 않는 폐지된 함수.

파이썬 3.6과 이전 버전에서는, 이 함수가 존재하지 않으면 GIL을 만들었습니다.

버전 3.9에서 변경: 이제 이 함수는 아무 작업도 수행하지 않습니다.

버전 3.7에서 변경: 이 함수는 이제 *`Py_Initialize()`*에 의해 호출되어서, 여러분은 더는 직접 호출할 필요가 없습니다.

버전 3.2에서 변경: 이 함수는 더는 *`Py_Initialize()`* 전에 호출할 수 없습니다.

버전 3.9부터 폐지됨.

PyThreadState ***PyEval_SaveThread**()

Part of the 안정 ABI. Detach the *attached thread state* and return it. The thread will have no *thread state* upon returning.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

Part of the 안정 ABI. Set the *attached thread state* to *tstate*. The passed *thread state* **should not** be *attached*, otherwise deadlock ensues. *tstate* will be attached upon returning.

> **참고**
>
> Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to *Cautions regarding runtime finalization* for more details.

버전 3.14에서 변경: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

PyThreadState ***PyThreadState_Get**()

Part of the 안정 ABI. Return the *attached thread state*. If the thread has no attached thread state, (such as when inside of `Py_BEGIN_ALLOW_THREADS` block), then this issues a fatal error (so that the caller needn' t check for `NULL`).

See also `PyThreadState_GetUnchecked()`.

PyThreadState ***PyThreadState_GetUnchecked**()

Similar to `PyThreadState_Get()`, but don' t kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL.

Added in version 3.13: In Python 3.5 to 3.12, the function was private and known as `_PyThreadState_UncheckedGet()`.

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

Part of the 안정 ABI. Set the *attached thread state* to *tstate*, and return the *thread state* that was attached prior to calling.

This function is safe to call without an *attached thread state*; it will simply return `NULL` indicating that there was no prior thread state.

> **참고**
>
> Similar to `PyGILState_Ensure()`, this function will hang the thread if the runtime is finalizing.

다음 함수는 스레드 로컬 저장소를 사용하며, 서브 인터프리터와 호환되지 않습니다:

PyGILState_STATE **PyGILState_Ensure**()

Part of the 안정 ABI. Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the *attached thread state*. This may be called as many times as desired by a thread as long as each call is matched with a call to `PyGILState_Release()`. In general, other thread-related APIs may be used between `PyGILState_Ensure()` and `PyGILState_Release()` calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros is acceptable.

The return value is an opaque "handle" to the *attached thread state* when `PyGILState_Ensure()` was called, and must be passed to `PyGILState_Release()` to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to `PyGILState_Ensure()` must save the handle for its call to `PyGILState_Release()`.

When the function returns, there will be an *attached thread state* and the thread will be able to call arbitrary Python code. Failure is a fatal error.

> **경고**
>
> Calling this function when the runtime is finalizing is unsafe. Doing so will either hang the thread until the program ends, or fully crash the interpreter in rare cases. Refer to *Cautions regarding runtime finalization* for more details.

버전 3.14에서 변경: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

void **PyGILState_Release**(PyGILState_STATE)

Part of the 안정 ABI. 이전에 획득 한 모든 자원을 해제합니다. 이 호출 후에, 파이썬의 상태는 해당 `PyGILState_Ensure()` 호출 이전과 같습니다 (그러나 일반적으로 이 상태는 호출자에게 알려지지 않아서, GILState API를 사용합니다).

`PyGILState_Ensure()`에 대한 모든 호출은 같은 스레드에서 `PyGILState_Release()`에 대한 호출과 쌍을 이뤄야 합니다.

PyThreadState ***PyGILState_GetThisThreadState**()

Part of the 안정 ABI. Get the *attached thread state* for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

> **참고**
>
> This function does not account for *thread states* created by something other than `PyGILState_Ensure()` (such as `PyThreadState_New()`). Prefer `PyThreadState_Get()` or `PyThreadState_GetUnchecked()` for most cases.

int **PyGILState_Check**()

Return `1` if the current thread is holding the *GIL* and `0` otherwise. This function can be called from any thread at any time. Only if it has had its *thread state* initialized via `PyGILState_Ensure()` will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the *GIL* is locked can allow the caller to perform sensitive actions or otherwise behave differently.

> **참고**
>
> If the current Python process has ever created a subinterpreter, this function will *always* return `1`. Prefer `PyThreadState_GetUnchecked()` for most cases.

Added in version 3.4.

다음 매크로는 일반적으로 후행 세미콜론 없이 사용됩니다; 파이썬 소스 배포에서 사용 예를 찾으십시오.

Py_BEGIN_ALLOW_THREADS

Part of the 안정 ABI. 이 매크로는 `{ PyThreadState *_save; _save = PyEval_SaveThread();`로 확장됩니다. 여는 중괄호가 포함되어 있음에 유의하십시오; 뒤따르는 `Py_END_ALLOW_THREADS` 매크로와 일치해야 합니다. 이 매크로에 대한 자세한 내용은 위를 참조하십시오.

Py_END_ALLOW_THREADS

Part of the 안정 ABI. 이 매크로는 `PyEval_RestoreThread(_save); }`로 확장됩니다. 닫는 중괄호가 포함되어 있음에 유의하십시오; 이전 `Py_BEGIN_ALLOW_THREADS` 매크로와 일치해야 합니다. 이 매크로에 대한 자세한 내용은 위를 참조하십시오.

Py_BLOCK_THREADS

Part of the 안정 ABI. 이 매크로는 `PyEval_RestoreThread(_save);`로 확장됩니다: 닫는 중괄호가 없는 `Py_END_ALLOW_THREADS`와 동등합니다.

Py_UNBLOCK_THREADS

Part of the 안정 ABI. 이 매크로는 `_save = PyEval_SaveThread();`로 확장됩니다: 여는 중괄호와 변수 선언이 없는 `Py_BEGIN_ALLOW_THREADS`와 동등합니다.

10.5.7 저수준 API

다음 함수는 모두 *Py_Initialize()* 이후에 호출되어야 합니다.

버전 3.7에서 변경: *Py_Initialize()* now initializes the *GIL* and sets an *attached thread state*.

PyInterpreterState ***PyInterpreterState_New**()

Part of the 안정 ABI. Create a new interpreter state object. An *attached thread state* is not needed, but may optionally exist if it is necessary to serialize calls to this function.

인자 없이 감사 이벤트 `cpython.PyInterpreterState_New`를 발생시킵니다.

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

Part of the 안정 ABI. Reset all information in an interpreter state object. There must be an *attached thread state* for the interpreter.

인자 없이 감사 이벤트 `cpython.PyInterpreterState_Clear`를 발생시킵니다.

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

Part of the 안정 ABI. Destroy an interpreter state object. There **should not** be an *attached thread state* for the target interpreter. The interpreter state must have been reset with a previous call to *PyInterpreterState_Clear()*.

PyThreadState ***PyThreadState_New**(*PyInterpreterState* *interp)

Part of the 안정 ABI. Create a new thread state object belonging to the given interpreter object. An *attached thread state* is not needed.

void **PyThreadState_Clear**(*PyThreadState* *tstate)

Part of the 안정 ABI. Reset all information in a *thread state* object. *tstate* must be *attached*

버전 3.9에서 변경: 이 함수는 이제 `PyThreadState.on_delete` 콜백을 호출합니다. 이전에는, *PyThreadState_Delete()*에서 호출했습니다.

버전 3.13에서 변경: The `PyThreadState.on_delete` callback was removed.

void **PyThreadState_Delete**(*PyThreadState* *tstate)

Part of the 안정 ABI. Destroy a *thread state* object. *tstate* should not be *attached* to any thread. *tstate* must have been reset with a previous call to *PyThreadState_Clear()*.

void **PyThreadState_DeleteCurrent**(void)

Detach the *attached thread state* (which must have been reset with a previous call to *PyThreadState_Clear()*) and then destroy it.

No *thread state* will be *attached* upon returning.

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

Part of the 안정 ABI 버전 *3.10* 이후로. 파이썬 스레드 상태 *tstate*의 현재 프레임을 가져옵니다.

Return a *strong reference*. Return `NULL` if no frame is currently executing.

*PyEval_GetFrame()*도 참조하십시오.

tstate must not be `NULL`, and must be *attached*.

Added in version 3.9.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

Part of the 안정 ABI 버전 *3.10* 이후로. Get the unique *thread state* identifier of the Python thread state *tstate*.

tstate must not be `NULL`, and must be *attached*.

Added in version 3.9.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

Part of the 안정 ABI 버전 *3.10* 이후로. 파이썬 스레드 상태 *tstate*의 인터프리터를 가져옵니다.

tstate must not be `NULL`, and must be *attached*.

Added in version 3.9.

void **PyThreadState_EnterTracing**(*PyThreadState* *tstate)

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the `PyThreadState_LeaveTracing()` function.

Added in version 3.11.

void **PyThreadState_LeaveTracing**(*PyThreadState* *tstate)

Resume tracing and profiling in the Python thread state *tstate* suspended by the `PyThreadState_EnterTracing()` function.

See also `PyEval_SetTrace()` and `PyEval_SetProfile()` functions.

Added in version 3.11.

PyInterpreterState ***PyInterpreterState_Get**(void)

Part of the 안정 ABI 버전 *3.9* 이후로. 현재 인터프리터를 가져옵니다.

Issue a fatal error if there no *attached thread state*. It cannot return NULL.

Added in version 3.9.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

Part of the 안정 ABI 버전 *3.7* 이후로. 인터프리터의 고유 ID를 반환합니다. 그렇게 하는데 에러가 발생하면 `-1`이 반환되고 에러가 설정됩니다.

The caller must have an *attached thread state*.

Added in version 3.7.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

반환값: 빌린 참조. *Part of the* 안정 ABI 버전 *3.8* 이후로. 인터프리터별 데이터가 저장될 수 있는 딕셔너리를 반환합니다. 이 함수가 `NULL`을 반환하면 예외는 발생하지 않았고 호출자는 인터프리터별 딕셔너리를 사용할 수 없다고 가정해야 합니다.

이것은 확장이 인터프리터별 상태 정보를 저장하는 데 사용해야 하는 `PyModule_GetState()`를 대체하는 것이 아닙니다.

The returned dictionary is borrowed from the interpreter and is valid until interpreter shutdown.

Added in version 3.8.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, *_PyInterpreterFrame* *frame, int throwflag)

프레임 평가 함수의 형.

throwflag 매개 변수는 제너레이터의 `throw()` 메서드에서 사용됩니다: 0이 아니면, 현재 예외를 처리합니다.

버전 3.9에서 변경: 이제 함수는 *tstate* 매개 변수를 취합니다.

버전 3.11에서 변경: The *frame* parameter changed from `PyFrameObject*` to `_PyInterpreterFrame*`.

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* *interp)

프레임 평가 함수를 가져옵니다.

PEP 523 "CPython에 프레임 평가 API 추가"를 참조하십시오.

Added in version 3.9.

void **_PyInterpreterState_SetEvalFrameFunc**(*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

프레임 평가 함수를 설정합니다.

PEP 523 "CPython에 프레임 평가 API 추가"를 참조하십시오.

Added in version 3.9.

PyObject ***PyThreadState_GetDict**()

반환값: 빌린 참조. *Part of the* 안정 ABI. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no *thread state* is *attached*. If this function returns `NULL`, no exception has been raised and the caller should assume no thread state is attached.

int **PyThreadState_SetAsyncExc**(unsigned long id, *PyObject* *exc)

Part of the 안정 ABI. Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with an *attached thread state*. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is `NULL`, the pending exception (if any) for the thread is cleared. This raises no exceptions.

버전 3.7에서 변경: The type of the *id* parameter changed from `long` to `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

Part of the 안정 ABI. *Attach tstate* to the current thread, which must not be `NULL` or already *attached*.

The calling thread must not already have an *attached thread state*.

> **참고**
>
> Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to *Cautions regarding runtime finalization* for more details.

버전 3.8에서 변경: *PyEval_RestoreThread()*, *Py_END_ALLOW_THREADS()* 및 *PyGILState_Ensure()*와 일관되도록 갱신되었으며, 인터프리터가 파이널리제이션 하는 동안 호출되면 현재 스레드를 종료합니다.

버전 3.14에서 변경: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

*PyEval_RestoreThread()*는 (스레드가 초기화되지 않았을 때조차) 항상 사용할 수 있는 고수준 함수입니다.

void **PyEval_ReleaseThread**(*PyThreadState* *tstate)

Part of the 안정 ABI. Detach the *attached thread state*. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the *attached thread state* — if it isn't, a fatal error is reported.

*PyEval_SaveThread()*는 (스레드가 초기화되지 않은 경우에조차) 항상 사용할 수 있는 고수준 함수입니다.

10.6 서브 인터프리터 지원

대부분의 경우, 단일 파이썬 인터프리터만 내장할 것입니다만, 같은 프로세스, 어쩌면 같은 스레드에서 여러 독립 인터프리터를 만들어야 하는 경우가 있습니다. 서브 인터프리터는 그렇게 할 수 있도록 합니다.

"메인" 인터프리터는 런타임이 초기화될 때 만들어지는 첫 번째 인터프리터입니다. 보통은 프로세스에서 유일한 파이썬 인터프리터입니다. 서브 인터프리터와 달리, 메인 인터프리터는 시그널 처리와 같은 고유한 프로세스 전역 책임을 갖습니다. 또한 런타임 초기화 동안 실행을 담당하며 일반적으로 런타임 파이널리제이션 동안 활성 인터프리터입니다. *PyInterpreterState_Main()* 함수는 그것의 상태에 대한 포인터를 반환합니다.

PyThreadState_Swap() 함수를 사용하여 서브 인터프리터 간에 전환할 수 있습니다. 다음 함수를 사용하여 만들고 파괴할 수 있습니다:

type **PyInterpreterConfig**

Structure containing most parameters to configure a sub-interpreter. Its values are used only in *Py_NewInterpreterFromConfig()* and never modified by the runtime.

Added in version 3.12.

Structure fields:

int **use_main_obmalloc**

If this is `0` then the sub-interpreter will use its own "object" allocator state. Otherwise it will use (share) the main interpreter' s.

If this is `0` then *check_multi_interp_extensions* must be `1` (non-zero). If this is `1` then *gil* must not be *PyInterpreterConfig_OWN_GIL*.

int **allow_fork**

If this is `0` then the runtime will not support forking the process in any thread where the sub-interpreter is currently active. Otherwise fork is unrestricted.

Note that the `subprocess` module still works when fork is disallowed.

int **allow_exec**

If this is `0` then the runtime will not support replacing the current process via exec (e.g. `os.execv()`) in any thread where the sub-interpreter is currently active. Otherwise exec is unrestricted.

Note that the `subprocess` module still works when exec is disallowed.

int **allow_threads**

If this is `0` then the sub-interpreter' s `threading` module won' t create threads. Otherwise threads are allowed.

int **allow_daemon_threads**

If this is `0` then the sub-interpreter' s `threading` module won' t create daemon threads. Otherwise daemon threads are allowed (as long as *allow_threads* is non-zero).

int **check_multi_interp_extensions**

If this is `0` then all extension modules may be imported, including legacy (single-phase init) modules, in any thread where the sub-interpreter is currently active. Otherwise only multi-phase init extension modules (see **PEP 489**) may be imported. (Also see *Py_mod_multiple_interpreters*.)

This must be `1` (non-zero) if *use_main_obmalloc* is `0`.

int **gil**

This determines the operation of the GIL for the sub-interpreter. It may be one of the following:

PyInterpreterConfig_DEFAULT_GIL

Use the default selection (*PyInterpreterConfig_SHARED_GIL*).

PyInterpreterConfig_SHARED_GIL

Use (share) the main interpreter' s GIL.

PyInterpreterConfig_OWN_GIL

Use the sub-interpreter' s own GIL.

If this is *PyInterpreterConfig_OWN_GIL* then *PyInterpreterConfig.use_main_obmalloc* must be `0`.

PyStatus **Py_NewInterpreterFromConfig**(*PyThreadState* **tstate_p, const *PyInterpreterConfig* *config)

새 서브 인터프리터를 만듭니다. 이것은 파이썬 코드 실행을 위한 (거의) 완전히 분리된 환경입니다. 특히, 새 인터프리터에는 기본 모듈 `builtins`, `__main__` 및 `sys`를 포함하여, 모든 임포트 된 모듈의 개별, 독립 버전을 갖습니다. 로드된 모듈 테이블(`sys.modules`)과 모듈 검색 경로(`sys.path`)도 별개입니다. 새 환경에는 `sys.argv` 변수가 없습니다. 새로운 표준 I/O 스트림 파일 객체 `sys.stdin`, `sys.stdout` 및 `sys.stderr`을 갖습니다 (단, 같은 하부 파일 기술자를 참조합니다).

The given *config* controls the options with which the interpreter is initialized.

Upon success, *tstate_p* will be set to the first *thread state* created in the new sub-interpreter. This thread state is *attached*. Note that no actual thread is created; see the discussion of thread states below. If creation of the

new interpreter is unsuccessful, *tstate_p* is set to `NULL`; no exception is set since the exception state is stored in the *attached thread state*, which might not exist.

Like all other Python/C API functions, an *attached thread state* must be present before calling this function, but it might be detached upon returning. On success, the returned thread state will be *attached*. If the sub-interpreter is created with its own *GIL* then the *attached thread state* of the calling interpreter will be detached. When the function returns, the new interpreter' s *thread state* will be *attached* to the current thread and the previous interpreter' s *attached thread state* will remain detached.

Added in version 3.12.

Sub-interpreters are most effective when isolated from each other, with certain functionality restricted:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

Note that the config is used only briefly and does not get modified. During initialization the config' s values are converted into various `PyInterpreterState` values. A read-only copy of the config may be stored internally on the `PyInterpreterState`.

확장 모듈은 다음과 같이 (서브) 인터프리터 간에 공유됩니다:

- 다단계 초기화를 사용하는 모듈의 경우, 예를 들어 `PyModule_FromDefAndSpec()`, 각 인터프리터에 대해 별도의 모듈 객체가 만들어지고 초기화됩니다. C 수준 정적과 전역 변수만 이러한 모듈 객체 간에 공유됩니다.
- 단단계 초기화를 사용하는 모듈의 경우, 예를 들어 `PyModule_Create()`, 특정 확장이 처음 임포트 될 때, 정상적으로 초기화되고, 모듈 딕셔너리의 (얕은) 사본이 저장됩니다. 다른 (서브) 인터프리터가 같은 확장을 임포트 할 때, 새 모듈이 초기화되고 이 복사본의 내용으로 채워집니다; 확장의 `init` 함수는 호출되지 않습니다. 따라서 모듈 딕셔너리의 객체는 (서브) 인터프리터 간에 공유되어, 원치 않는 동작을 일으킬 수 있습니다 (아래 버그와 주의 사항을 참조하십시오).

  이것은 인터프리터가 `Py_FinalizeEx()`와 `Py_Initialize()`를 호출하여 완전히 다시 초기화된 후 확장을 임포트 할 때 일어나는 것과 다름에 유의하십시오; 이 경우, 확장의 `initmodule` 함수가 다시 호출됩니다. 다단계 초기화와 마찬가지로, 이는 C 수준의 정적과 전역 변수만 이러한 모듈 간에 공유됨을 의미합니다.

PyThreadState ***Py_NewInterpreter**(void)

Part of the 안정 ABI. Create a new sub-interpreter. This is essentially just a wrapper around `Py_NewInterpreterFromConfig()` with a config that preserves the existing behavior. The result is an unisolated sub-interpreter that shares the main interpreter' s GIL, allows fork/exec, allows daemon threads, and allows single-phase init modules.

void **Py_EndInterpreter**(*PyThreadState* *tstate)

Part of the 안정 ABI. Destroy the (sub-)interpreter represented by the given *thread state*. The given thread state must be *attached*. When the call returns, there will be no *attached thread state*. All thread states associated with this interpreter are destroyed.

`Py_FinalizeEx()` will destroy all sub-interpreters that haven' t been explicitly destroyed at that point.

10.6.1 A Per-Interpreter GIL

Using *Py_NewInterpreterFromConfig()* you can create a sub-interpreter that is completely isolated from other interpreters, including having its own GIL. The most important benefit of this isolation is that such an interpreter can execute Python code without being blocked by other interpreters or blocking any others. Thus a single Python process can truly take advantage of multiple CPU cores when running Python code. The isolation also encourages a different approach to concurrency than that of just using threads. (See **PEP 554** and **PEP 684**.)

Using an isolated interpreter requires vigilance in preserving that isolation. That especially means not sharing any objects or mutable state without guarantees about thread-safety. Even objects that are otherwise immutable (e.g. `None`, `(1, 5)`) can't normally be shared because of the refcount. One simple but less-efficient approach around this is to use a global lock around all use of some state (or object). Alternately, effectively immutable objects (like integers or strings) can be made safe in spite of their refcounts by making them *immortal*. In fact, this has been done for the builtin singletons, small integers, and a number of other builtin objects.

If you preserve isolation then you will have access to proper multi-core computing without the complications that come with free-threading. Failure to preserve isolation will expose you to the full consequences of free-threading, including races and hard-to-debug crashes.

Aside from that, one of the main challenges of using multiple isolated interpreters is how to communicate between them safely (not break isolation) and efficiently. The runtime and stdlib do not provide any standard approach to this yet. A future stdlib module would help mitigate the effort of preserving isolation and expose effective tools for communicating (and sharing) data between interpreters.

Added in version 3.12.

10.6.2 버그와 주의 사항

서브 인터프리터(및 메인 인터프리터)는 같은 프로세스의 일부이기 때문에, 그들 간의 절연이 완벽하지 않습니다 — 예를 들어, `os.close()`와 같은 저수준 파일 연산을 사용하면 서로의 열린 파일에 (실수로 혹은 악의적으로) 영향을 미칠 수 있습니다. (서브) 인터프리터 간에 확장이 공유되는 방식 때문에, 일부 확장이 제대로 작동하지 않을 수 있습니다; 이것은 특히 단단계 초기화나 (정적) 전역 변수를 사용할 때 특히 그렇습니다. 한 서브 인터프리터에서 만든 객체를 다른 (서브) 인터프리터의 이름 공간에 삽입할 수 있습니다; 가능하면 피해야 합니다.

서브 인터프리터 간에 사용자 정의 함수, 메서드, 인스턴스 또는 클래스를 공유하지 않도록 특별한 주의를 기울여야 합니다. 이러한 객체에 의해 실행되는 임포트 연산은 잘못된 (서브) 인터프리터의 로드된 모듈 딕셔너리에 영향을 미칠 수 있기 때문입니다. 위의 것들에서 접근할 수 있는 객체를 공유하지 않는 것도 마찬가지로 중요합니다.

Also note that combining this functionality with `PyGILState_*` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching *PyGILState_Ensure()* and *PyGILState_Release()* calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

10.7 비동기 알림

메인 인터프리터 스레드에 비동기 알림을 보내는 메커니즘이 제공됩니다. 이러한 알림은 함수 포인터와 void 포인터 인자의 형태를 취합니다.

int **Py_AddPendingCall**(int (*func)(void*), void *arg)

Part of the 안정 ABI. 메인 인터프리터 스레드에서 호출할 함수를 예약합니다. 성공하면 `0`이 반환되고 *func*는 메인 스레드에서 호출되기 위해 큐에 추가됩니다. 실패 시, 예외 설정 없이 `-1`이 반환됩니다.

성공적으로 큐에 넣으면, *func*는 *arg* 인자를 사용하여 결국 메인 인터프리터 스레드에서 호출됩니다. 정상적으로 실행되는 파이썬 코드와 비교할 때 비동기적으로 호출되지만, 다음 두 조건이 모두 충족됩니다:

- 바이트 코드 경계에서;
- with the main thread holding an *attached thread state* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the *thread state* is detached.

This function doesn't need an *attached thread state*. However, to call this function in a subinterpreter, the caller must have an *attached thread state*. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

> **경고**
>
> 이것은 매우 특별한 경우에만 유용한, 저수준 함수입니다. *func*가 가능한 한 빨리 호출된다는 보장은 없습니다. 메인 스레드가 시스템 호출을 실행 중이라 바쁘면, 시스템 호출이 반환되기 전에 *func*가 호출되지 않습니다. 이 함수는 일반적으로 임의의 C 스레드에서 파이썬 코드를 호출하는 데 적합하지 **않습니다**. 대신, *PyGILState API*를 사용하십시오.

Added in version 3.1.

버전 3.9에서 변경: 이 함수가 서브 인터프리터에서 호출되면, *func* 함수는 이제 메인 인터프리터에서 호출되지 않고 서브 인터프리터에서 호출되도록 예약됩니다. 이제 각 서브 인터프리터는 자체 예약된 호출 목록을 갖습니다.

10.8 프로파일링과 추적

파이썬 인터프리터는 프로파일링과 실행 추적 기능을 연결하기 위한 몇 가지 저수준 지원을 제공합니다. 프로파일링, 디버깅 및 커버리지 분석 도구에 사용됩니다.

이 C 인터페이스를 사용하면 프로파일링이나 추적 코드가 파이썬 수준의 콜러블 객체를 통해 호출하는 오버헤드를 피하고, 대신 직접 C 함수를 호출할 수 있습니다. 시설의 필수 어트리뷰트는 변경되지 않았습니다; 인터페이스는 추적 함수를 스레드별로 설치할 수 있도록 하며, 추적 함수에 보고되는 기본 이벤트는 이전 버전의 파이썬 수준 추적 함수에 보고된 것과 같습니다.

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

*what*의 값	*arg*의 의미
`PyTrace_CALL`	항상 `Py_None`.
`PyTrace_EXCEPTION`	`sys.exc_info()`에서 반환된 예외 정보.
`PyTrace_LINE`	항상 `Py_None`.
`PyTrace_RETURN`	호출자에게 반환되는 값, 또는 예외로 인한 것이면 `NULL`.
`PyTrace_C_CALL`	호출되는 함수 객체.
`PyTrace_C_EXCEPTION`	호출되는 함수 객체.
`PyTrace_C_RETURN`	호출되는 함수 객체.
`PyTrace_OPCODE`	항상 `Py_None`.

int **PyTrace_CALL**

함수나 메서드에 대한 새 호출이 보고되거나, 제너레이터에 대한 새 항목이 보고될 때 `Py_tracefunc` 함수에 대한 *what* 매개 변수의 값. 제너레이터 함수에 대한 이터레이터의 생성은 해당 프레임의 파이썬 바이트 코드로의 제어 전송이 없기 때문에 보고되지 않음에 유의하십시오.

int **PyTrace_EXCEPTION**

예외가 발생했을 때 `Py_tracefunc` 함수에 대한 *what* 매개 변수의 값. 콜백 함수는 실행되는 프레임 내에서 바이트 코드가 처리된 후 예외가 설정될 때 *what*에 대해 이 값으로 호출됩니다. 이것의 효과

는 예외 전파로 인해 파이썬 스택이 되감기는 것입니다, 예외가 전파되어 각 프레임으로 반환할 때 콜백이 호출됩니다. 추적 함수만 이러한 이벤트를 수신합니다; 프로파일러에는 필요하지 않습니다.

int **PyTrace_LINE**

The value passed as the *what* parameter to a *Py_tracefunc* function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int **PyTrace_RETURN**

호출이 반환되려고 할 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int **PyTrace_C_CALL**

C 함수가 호출되려고 할 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int **PyTrace_C_EXCEPTION**

C 함수에서 예외가 발생했을 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int **PyTrace_C_RETURN**

C 함수가 반환했을 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int **PyTrace_OPCODE**

The value for the *what* parameter to *Py_tracefunc* functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile**(*Py_tracefunc* func, *PyObject* *obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except *PyTrace_LINE* *PyTrace_OPCODE* and *PyTrace_EXCEPTION*.

See also the `sys.setprofile()` function.

The caller must have an *attached thread state*.

void **PyEval_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetProfile()* but sets the profile function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an *attached thread state*.

As *PyEval_SetProfile()*, this function ignores any exceptions raised while setting the profile functions in all threads.

Added in version 3.12.

void **PyEval_SetTrace**(*Py_tracefunc* func, *PyObject* *obj)

Set the tracing function to *func*. This is similar to *PyEval_SetProfile()*, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using *PyEval_SetTrace()* will not receive *PyTrace_C_CALL*, *PyTrace_C_EXCEPTION* or *PyTrace_C_RETURN* as a value for the *what* parameter.

See also the `sys.settrace()` function.

The caller must have an *attached thread state*.

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetTrace()* but sets the tracing function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an *attached thread state*.

As *PyEval_SetTrace()*, this function ignores any exceptions raised while setting the trace functions in all threads.

Added in version 3.12.

10.9 Reference tracing

Added in version 3.13.

typedef int (*__PyRefTracer__)(*PyObject**, int event, void *data)

The type of the trace function registered using `PyRefTracer_SetTracer()`. The first parameter is a Python object that has been just created (when **event** is set to `PyRefTracer_CREATE`) or about to be destroyed (when **event** is set to `PyRefTracer_DESTROY`). The **data** argument is the opaque pointer that was provided when `PyRefTracer_SetTracer()` was called.

Added in version 3.13.

int **PyRefTracer_CREATE**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been created.

int **PyRefTracer_DESTROY**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been destroyed.

int **PyRefTracer_SetTracer**(*PyRefTracer* tracer, void *data)

Register a reference tracer function. The function will be called when a new Python has been created or when an object is going to be destroyed. If **data** is provided it must be an opaque pointer that will be provided when the tracer function is called. Return `0` on success. Set an exception and return `-1` on error.

Not that tracer functions **must not** create Python objects inside or otherwise the call will be re-entrant. The tracer also **must not** clear any existing exception or set an exception. A *thread state* will be active every time the tracer function is called.

There must be an *attached thread state* when calling this function.

Added in version 3.13.

PyRefTracer **PyRefTracer_GetTracer**(void **data)

Get the registered reference tracer function and the value of the opaque data pointer that was registered when `PyRefTracer_SetTracer()` was called. If no tracer was registered this function will return NULL and will set the **data** pointer to NULL.

There must be an *attached thread state* when calling this function.

Added in version 3.13.

10.10 고급 디버거 지원

이 함수들은 고급 디버깅 도구에서만 사용하기 위한 것입니다.

PyInterpreterState ***PyInterpreterState_Head**()

인터프리터 상태 객체들의 리스트의 머리에 있는 객체를 반환합니다.

PyInterpreterState ***PyInterpreterState_Main**()

메인 인터프리터 상태 객체를 반환합니다.

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

인터프리터 상태 객체들의 리스트에서 *interp* 이후의 다음 인터프리터 상태 객체를 반환합니다.

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

인터프리터 *interp*와 관련된 스레드 리스트에서 첫 번째 `PyThreadState` 객체에 대한 포인터를 반환합니다.

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

같은 `PyInterpreterState` 객체에 속하는 모든 스레드 객체 리스트에서 *tstate* 이후의 다음 스레드 상태 객체를 반환합니다.

10.11 스레드 로컬 저장소 지원

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

A *thread state* does *not* need to be *attached* when calling these functions; they suppl their own locking.

`Python.h`에는 TLS API 선언이 포함되어 있지 않음에 유의하십시오, 스레드 로컬 저장소를 사용하려면 `pythread.h`를 포함해야 합니다.

참고

None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be *`PyObject`*`*`, these functions don' t do refcount operations on them either.

10.11.1 스레드별 저장소 (TSS - Thread Specific Storage) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type *`Py_tss_t`* instead of `int` to represent thread keys.

Added in version 3.7.

더 보기

"CPython의 스레드-로컬 저장소를 위한 새로운 C-API" (**PEP 539**)

type **`Py_tss_t`**

이 자료 구조는 스레드 키의 상태를 나타내며, 정의는 하부 TLS 구현에 따라 달라질 수 있으며, 키의 초기화 상태를 나타내는 내부 필드가 있습니다. 이 구조체에는 공개 멤버가 없습니다.

*Py_LIMITED_API*가 정의되지 않을 때, *`Py_tss_NEEDS_INIT`*로 이 형의 정적 할당이 허용됩니다.

`Py_tss_NEEDS_INIT`

이 매크로는 *`Py_tss_t`* 변수의 초기화자(initializer)로 확장됩니다. 이 매크로는 *Py_LIMITED_API*에서 정의되지 않음에 유의하십시오.

동적 할당

*Py_LIMITED_API*로 빌드된 확장 모듈에 필요한, 빌드 시점에 구현이 불투명해서 형의 정적 할당이 불가능한 *`Py_tss_t`*의 동적 할당.

Py_tss_t ***`PyThread_tss_alloc`**`()`

Part of the 안정 ABI 버전 *3.7* 이후로. *`Py_tss_NEEDS_INIT`*로 초기화된 값과 같은 상태의 값을 반환하거나, 동적 할당 실패 시 `NULL`을 반환합니다.

void **`PyThread_tss_free`**`(`*Py_tss_t* *key`)`

Part of the 안정 ABI 버전 *3.7* 이후로. Free the given *key* allocated by *`PyThread_tss_alloc()`*, after first calling *`PyThread_tss_delete()`* to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is `NULL`.

참고

A freed key becomes a dangling pointer. You should reset the key to `NULL`.

메서드

이 함수들의 매개 변수 *key*는 `NULL`이 아니어야 합니다. 또한, 주어진 *Py_tss_t*가 *PyThread_tss_create()*로 초기화되지 않았으면, *PyThread_tss_set()*과 *PyThread_tss_get()*의 동작은 정의되지 않습니다.

int **PyThread_tss_is_created**(*Py_tss_t* *key)

Part of the 안정 ABI 버전 *3.7* 이후로. 주어진 *Py_tss_t*가 *PyThread_tss_create()*로 초기화되었으면 0이 아닌 값을 반환합니다.

int **PyThread_tss_create**(*Py_tss_t* *key)

Part of the 안정 ABI 버전 *3.7* 이후로. TSS 키 초기화에 성공하면 0 값을 반환합니다. *key* 인자가 가리키는 값이 *Py_tss_NEEDS_INIT*로 초기화되지 않으면 동작이 정의되지 않습니다. 이 함수는 같은 키에서 반복적으로 호출될 수 있습니다 – 이미 초기화된 키에 대해 호출하면 아무런 일도 하지 않으며 즉시 성공을 반환합니다.

void **PyThread_tss_delete**(*Py_tss_t* *key)

Part of the 안정 ABI 버전 *3.7* 이후로. TSS 키를 삭제하여 모든 스레드에서 키와 관련된 값을 잊게 하고, 키의 초기화 상태를 초기화되지 않음으로 변경합니다. 파괴된 키는 *PyThread_tss_create()*로 다시 초기화할 수 있습니다. 이 함수는 같은 키에서 반복적으로 호출될 수 있습니다 – 이미 파괴된 키에 대해 호출하면 아무런 일도 하지 않습니다.

int **PyThread_tss_set**(*Py_tss_t* *key, void *value)

Part of the 안정 ABI 버전 *3.7* 이후로. Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void ***PyThread_tss_get**(*Py_tss_t* *key)

Part of the 안정 ABI 버전 *3.7* 이후로. Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

10.11.2 스레드 로컬 저장소 (TLS) API

버전 3.7부터 폐지됨: 이 API는 스레드별 저장소 (*TSS*) *API*로 대체됩니다.

> **참고**
>
> 이 버전의 API는 `int`로 안전하게 캐스트 할 수 없는 방식으로 네이티브 TLS 키가 정의된 플랫폼을 지원하지 않습니다. 이러한 플랫폼에서, *PyThread_create_key()*는 실패 상태로 즉시 반환되며, 다른 TLS 함수는 이러한 플랫폼에서 모두 아무런 일도 하지 않습니다.

위에서 언급한 호환성 문제로 인해, 이 버전의 API를 새 코드에서 사용해서는 안 됩니다.

int **PyThread_create_key**()

Part of the 안정 ABI.

void **PyThread_delete_key**(int key)

Part of the 안정 ABI.

int **PyThread_set_key_value**(int key, void *value)

Part of the 안정 ABI.

void ***PyThread_get_key_value**(int key)

Part of the 안정 ABI.

void **PyThread_delete_key_value**(int key)

Part of the 안정 ABI.

void **PyThread_ReInitTLS**()

Part of the 안정 ABI.

10.12 Synchronization Primitives

The C-API provides a basic mutual exclusion lock.

type **PyMutex**

A mutual exclusion lock. The `PyMutex` should be initialized to zero to represent the unlocked state. For example:

```
PyMutex mutex = {0};
```

Instances of `PyMutex` should not be copied or moved. Both the contents and address of a `PyMutex` are meaningful, and it must remain at a fixed, writable location in memory.

> **참고**
>
> A `PyMutex` currently occupies one byte, but the size should be considered unstable. The size may change in future Python releases without a deprecation period.

Added in version 3.13.

void **PyMutex_Lock**(*PyMutex* *m)

Lock mutex *m*. If another thread has already locked it, the calling thread will block until the mutex is unlocked. While blocked, the thread will temporarily detach the *thread state* if one exists.

Added in version 3.13.

void **PyMutex_Unlock**(*PyMutex* *m)

Unlock mutex *m*. The mutex must be locked — otherwise, the function will issue a fatal error.

Added in version 3.13.

int **PyMutex_IsLocked**(*PyMutex* *m)

Returns non-zero if the mutex *m* is currently locked, zero otherwise.

> **참고**
>
> This function is intended for use in assertions and debugging only and should not be used to make concurrency control decisions, as the lock state may change immediately after the check.

Added in version 3.14.

10.12.1 Python Critical Section API

The critical section API provides a deadlock avoidance layer on top of per-object locks for *free-threaded* CPython. They are intended to replace reliance on the *global interpreter lock*, and are no-ops in versions of Python with the global interpreter lock.

Critical sections are intended to be used for custom types implemented in C-API extensions. They should generally not be used with built-in types like `list` and `dict` because their public C-APIs already use critical sections internally, with the notable exception of `PyDict_Next()`, which requires critical section to be acquired externally.

Critical sections avoid deadlocks by implicitly suspending active critical sections, hence, they do not provide exclusive access such as provided by traditional locks like `PyMutex`. When a critical section is started, the per-object lock for the object is acquired. If the code executed inside the critical section calls C-API functions then it can suspend the critical section thereby releasing the per-object lock, so other threads can acquire the per-object lock for the same object.

Variants that accept `PyMutex` pointers rather than Python objects are also available. Use these variants to start a critical section in a situation where there is no `PyObject` – for example, when working with a C type that does not extend or wrap `PyObject` but still needs to call into the C API in a manner that might lead to deadlocks.

The functions and structs used by the macros are exposed for cases where C macros are not available. They should only be used as in the given macro expansions. Note that the sizes and contents of the structures may change in future Python versions.

> **참고**
>
> Operations that need to lock two objects at once must use *Py_BEGIN_CRITICAL_SECTION2*. You *cannot* use nested critical sections to lock more than one object at once, because the inner critical section may suspend the outer critical sections. This API does not provide a way to lock more than two objects at once.

Example usage:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

In the above example, *Py_SETREF* calls *Py_DECREF*, which can call arbitrary code through an object' s deallocation function. The critical section API avoids potential deadlocks due to reentrancy and lock ordering by allowing the runtime to temporarily suspend the critical section if the code triggered by the finalizer blocks and calls *PyEval_SaveThread()*.

Py_BEGIN_CRITICAL_SECTION(op)

Acquires the per-object lock for the object *op* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION_MUTEX(m)

Locks the mutex *m* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
     PyCriticalSection _py_cs;
     PyCriticalSection_BeginMutex(&_py_cs, m)
```

Note that unlike *Py_BEGIN_CRITICAL_SECTION*, there is no cast for the argument of the macro - it must be a *PyMutex* pointer.

On the default build, this macro expands to `{`.

Added in version 3.14.

Py_END_CRITICAL_SECTION()

Ends the critical section and releases the per-object lock.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection_End(&_py_cs);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2(a, b)

Acquires the per-objects locks for the objects *a* and *b* and begins a critical section. The locks are acquired in a consistent order (lowest address first) to avoid lock ordering deadlocks.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2_MUTEX(m1, m2)

Locks the mutexes *m1* and *m2* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
     PyCriticalSection2 _py_cs2;
     PyCriticalSection2_BeginMutex(&_py_cs2, m1, m2)
```

Note that unlike *Py_BEGIN_CRITICAL_SECTION2*, there is no cast for the arguments of the macro - they must be *PyMutex* pointers.

On the default build, this macro expands to `{`.

Added in version 3.14.

Py_END_CRITICAL_SECTION2()

Ends the critical section and releases the per-object locks.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection2_End(&_py_cs2);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

CHAPTER 11

파이썬 초기화 구성

11.1 PyInitConfig C API

Added in version 3.14.

Python can be initialized with *Py_InitializeFromInitConfig()*.

The *Py_RunMain()* function can be used to write a customized Python program.

초기화, 파이널리제이션 및 스레드도 참조하십시오.

> **더 보기**
>
> **PEP 741** "Python Configuration C API".

11.1.1 Example

Example of customized Python always running with the Python Development Mode enabled; return `-1` on error:

```
int init_python(void)
{
    PyInitConfig *config = PyInitConfig_Create();
    if (config == NULL) {
        printf("PYTHON INIT ERROR: memory allocation failed\n");
        return -1;
    }

    // Enable the Python Development Mode
    if (PyInitConfig_SetInt(config, "dev_mode", 1) < 0) {
        goto error;
    }

    // Initialize Python with the configuration
    if (Py_InitializeFromInitConfig(config) < 0) {
        goto error;
    }
    PyInitConfig_Free(config);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    return 0;

error:
    {
        // Display the error message.
        //
        // This uncommon braces style is used, because you cannot make
        // goto targets point to variable declarations.
        const char *err_msg;
        (void)PyInitConfig_GetError(config, &err_msg);
        printf("PYTHON INIT ERROR: %s\n", err_msg);
        PyInitConfig_Free(config);
        return -1;
    }
}
```

11.1.2 Create Config

struct **PyInitConfig**

Opaque structure to configure the Python initialization.

PyInitConfig ***PyInitConfig_Create**(void)

Create a new initialization configuration using *Isolated Configuration* default values.

It must be freed by *PyInitConfig_Free()*.

Return `NULL` on memory allocation failure.

void **PyInitConfig_Free**(*PyInitConfig* *config)

Free memory of the initialization configuration *config*.

If *config* is `NULL`, no operation is performed.

11.1.3 Error Handling

int **PyInitConfig_GetError**(*PyInitConfig* *config, const char **err_msg)

Get the *config* error message.

- Set **err_msg* and return `1` if an error is set.
- Set **err_msg* to `NULL` and return `0` otherwise.

An error message is an UTF-8 encoded string.

If *config* has an exit code, format the exit code as an error message.

The error message remains valid until another `PyInitConfig` function is called with *config*. The caller doesn't have to free the error message.

int **PyInitConfig_GetExitCode**(*PyInitConfig* *config, int *exitcode)

Get the *config* exit code.

- Set **exitcode* and return `1` if *config* has an exit code set.
- Return `0` if *config* has no exit code set.

Only the `Py_InitializeFromInitConfig()` function can set an exit code if the `parse_argv` option is non-zero.

An exit code can be set when parsing the command line failed (exit code `2`) or when a command line option asks to display the command line help (exit code `0`).

11.1.4 Get Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

int **PyInitConfig_HasOption**(*PyInitConfig* *config, const char *name)

Test if the configuration has an option called *name*.

Return `1` if the option exists, or return `0` otherwise.

int **PyInitConfig_GetInt**(*PyInitConfig* *config, const char *name, int64_t *value)

Get an integer configuration option.

- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_GetStr**(*PyInitConfig* *config, const char *name, char **value)

Get a string configuration option as a null-terminated UTF-8 encoded string.

- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

**value* can be set to `NULL` if the option is an optional string and the option is unset.

On success, the string must be released with `free(value)` if it' s not `NULL`.

int **PyInitConfig_GetStrList**(*PyInitConfig* *config, const char *name, size_t *length, char ***items)

Get a string list configuration option as an array of null-terminated UTF-8 encoded strings.

- Set **length* and **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

On success, the string list must be released with `PyInitConfig_FreeStrList(length, items)`.

void **PyInitConfig_FreeStrList**(size_t length, char **items)

Free memory of a string list created by `PyInitConfig_GetStrList()`.

11.1.5 Set Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

Some configuration options have side effects on other options. This logic is only implemented when `Py_InitializeFromInitConfig()` is called, not by the "Set" functions below. For example, setting `dev_mode` to `1` does not set `faulthandler` to `1`.

int **PyInitConfig_SetInt**(*PyInitConfig* *config, const char *name, int64_t value)

Set an integer configuration option.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_SetStr**(*PyInitConfig* *config, const char *name, const char *value)

Set a string configuration option from a null-terminated UTF-8 encoded string. The string is copied.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_SetStrList**(*PyInitConfig* *config, const char *name, size_t length, char *const *items)

Set a string list configuration option from an array of null-terminated UTF-8 encoded strings. The string list is copied.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

11.1.6 Module

int **PyInitConfig_AddModule**(*PyInitConfig* *config, const char *name, *PyObject* *(*initfunc)(void))

Add a built-in extension module to the table of built-in modules.

The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

If Python is initialized multiple times, `PyInitConfig_AddModule()` must be called at each Python initialization.

Similar to the *PyImport_AppendInittab()* function.

11.1.7 Initialize Python

int **Py_InitializeFromInitConfig**(*PyInitConfig* *config)

Initialize Python from the initialization configuration.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.
- Set an exit code in *config* and return `-1` if Python wants to exit.

See `PyInitConfig_GetExitcode()` for the exit code case.

11.2 Configuration Options

Option	PyConfig/PyPreConfig member	Type	Visibility
"allocator"	*allocator*	int	Read-only
"argv"	*argv*	list[str]	Public
"base_exec_prefix"	*base_exec_prefix*	str	Public
"base_executable"	*base_executable*	str	Public
"base_prefix"	*base_prefix*	str	Public
"buffered_stdio"	*buffered_stdio*	bool	Read-only
"bytes_warning"	*bytes_warning*	int	Public
"check_hash_pycs_mode"	*check_hash_pycs_mode*	str	Read-only
"code_debug_ranges"	*code_debug_ranges*	bool	Read-only
"coerce_c_locale"	*coerce_c_locale*	bool	Read-only
"coerce_c_locale_warn"	*coerce_c_locale_warn*	bool	Read-only
"configure_c_stdio"	*configure_c_stdio*	bool	Read-only
"configure_locale"	*configure_locale*	bool	Read-only
"cpu_count"	*cpu_count*	int	Public
"dev_mode"	*dev_mode*	bool	Read-only
"dump_refs"	*dump_refs*	bool	Read-only
"dump_refs_file"	*dump_refs_file*	str	Read-only
"exec_prefix"	*exec_prefix*	str	Public
"executable"	*executable*	str	Public
"faulthandler"	*faulthandler*	bool	Read-only
"filesystem_encoding"	*filesystem_encoding*	str	Read-only
"filesystem_errors"	*filesystem_errors*	str	Read-only
"hash_seed"	*hash_seed*	int	Read-only
"home"	*home*	str	Read-only
"import_time"	*import_time*	int	Read-only
"inspect"	*inspect*	bool	Public

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

Option	PyConfig/PyPreConfig member	Type	Visibility
"install_signal_handlers"	*install_signal_handlers*	bool	Read-only
"int_max_str_digits"	*int_max_str_digits*	int	Public
"interactive"	*interactive*	bool	Public
"isolated"	*isolated*	bool	Read-only
"legacy_windows_fs_encoding"	*legacy_windows_fs_encoding*	bool	Read-only
"legacy_windows_stdio"	*legacy_windows_stdio*	bool	Read-only
"malloc_stats"	*malloc_stats*	bool	Read-only
"module_search_paths"	*module_search_paths*	list[str]	Public
"optimization_level"	*optimization_level*	int	Public
"orig_argv"	*orig_argv*	list[str]	Read-only
"parse_argv"	*parse_argv*	bool	Read-only
"parser_debug"	*parser_debug*	bool	Public
"pathconfig_warnings"	*pathconfig_warnings*	bool	Read-only
"perf_profiling"	*perf_profiling*	bool	Read-only
"platlibdir"	*platlibdir*	str	Public
"prefix"	*prefix*	str	Public
"program_name"	*program_name*	str	Read-only
"pycache_prefix"	*pycache_prefix*	str	Public
"quiet"	*quiet*	bool	Public
"run_command"	*run_command*	str	Read-only
"run_filename"	*run_filename*	str	Read-only
"run_module"	*run_module*	str	Read-only
"run_presite"	*run_presite*	str	Read-only
"safe_path"	*safe_path*	bool	Read-only
"show_ref_count"	*show_ref_count*	bool	Read-only
"site_import"	*site_import*	bool	Read-only
"skip_source_first_line"	*skip_source_first_line*	bool	Read-only
"stdio_encoding"	*stdio_encoding*	str	Read-only
"stdio_errors"	*stdio_errors*	str	Read-only
"stdlib_dir"	*stdlib_dir*	str	Public
"tracemalloc"	*tracemalloc*	int	Read-only
"use_environment"	*use_environment*	bool	Public
"use_frozen_modules"	*use_frozen_modules*	bool	Read-only
"use_hash_seed"	*use_hash_seed*	bool	Read-only
"use_system_logger"	*use_system_logger*	bool	Read-only
"user_site_directory"	*user_site_directory*	bool	Read-only
"utf8_mode"	*utf8_mode*	bool	Read-only
"verbose"	*verbose*	int	Public
"warn_default_encoding"	*warn_default_encoding*	bool	Read-only
"warnoptions"	*warnoptions*	list[str]	Public
"write_bytecode"	*write_bytecode*	bool	Public
"xoptions"	*xoptions*	dict[str, str]	Public
"_pystats"	*_pystats*	bool	Read-only

Visibility:

- Public: Can by get by *PyConfig_Get()* and set by *PyConfig_Set()*.
- Read-only: Can by get by *PyConfig_Get()*, but cannot be set by *PyConfig_Set()*.

11.3 Runtime Python configuration API

At runtime, it' s possible to get and set configuration options using *PyConfig_Get()* and *PyConfig_Set()* functions.

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

Some options are read from the `sys` attributes. For example, the option `"argv"` is read from `sys.argv`.

PyObject ***PyConfig_Get**(const char *name)

Get the current runtime value of a configuration option as a Python object.

- Return a new reference on success.
- Set an exception and return `NULL` on error.

The object type depends on the configuration option. It can be:

- `bool`
- `int`
- `str`
- `list[str]`
- `dict[str, str]`

The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Added in version 3.14.

int **PyConfig_GetInt**(const char *name, int *value)

Similar to *PyConfig_Get()*, but get the value as a C int.

- Return `0` on success.
- Set an exception and return `-1` on error.

Added in version 3.14.

PyObject ***PyConfig_Names**(void)

Get all configuration option names as a `frozenset`.

- Return a new reference on success.
- Set an exception and return `NULL` on error.

The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Added in version 3.14.

int **PyConfig_Set**(const char *name, *PyObject* *value)

Set the current runtime value of a configuration option.

- Raise a `ValueError` if there is no option *name*.
- Raise a `ValueError` if *value* is an invalid value.
- Raise a `ValueError` if the option is read-only (cannot be set).
- Raise a `TypeError` if *value* has not the proper type.

The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Raises an auditing event `cpython.PyConfig_Set` with arguments `name`, `value`.

Added in version 3.14.

11.4 PyConfig C API

Added in version 3.8.

Python can be initialized with `Py_InitializeFromConfig()` and the `PyConfig` structure. It can be preinitialized with `Py_PreInitialize()` and the `PyPreConfig` structure.

There are two kinds of configuration:

- The *Python Configuration* can be used to build a customized Python which behaves as the regular Python. For example, environment variables and command line arguments are used to configure Python.
- The *Isolated Configuration* can be used to embed Python into an application. It isolates Python from the system. For example, environment variables are ignored, the LC_CTYPE locale is left unchanged and no signal handler is registered.

The `Py_RunMain()` function can be used to write a customized Python program.

초기화, 파이널리제이션 및 스레드도 참조하십시오.

더 보기

PEP 587 "파이썬 초기화 구성".

11.4.1 Example

항상 격리 모드에서 실행되는 사용자 정의 파이썬의 예:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

11.4.2 PyWideStringList

type **PyWideStringList**

`wchar_t*` 문자열의 리스트.

*length*가 0이 아니면, *items*는 `NULL`이 아니어야 하고 모든 문자열은 `NULL`이 아니어야 합니다.

메서드:

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

*item*을 *list*에 추가합니다.

이 함수를 호출하려면 파이썬을 사전 초기화해야 합니다.

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

*item*을 *list*의 *index*에 삽입합니다.

*index*가 *list* 길이보다 크거나 같으면, *item*을 *list*에 추가(append)합니다.

index must be greater than or equal to `0`.

이 함수를 호출하려면 파이썬을 사전 초기화해야 합니다.

구조체 필드:

Py_ssize_t **length**

리스트 길이.

wchar_t ***items**

리스트 항목들.

11.4.3 PyStatus

type **PyStatus**

초기화 함수 상태를 저장하는 구조체: 성공, 에러 또는 종료.

에러의 경우, 에러를 만든 C 함수 이름을 저장할 수 있습니다.

구조체 필드:

int **exitcode**

종료 코드. `exit()`에 전달된 인자.

const char ***err_msg**

에러 메시지.

const char ***func**

에러를 만든 함수의 이름, `NULL`일 수 있습니다.

상태를 만드는 함수:

PyStatus **PyStatus_Ok**(void)

성공.

PyStatus **PyStatus_Error**(const char *err_msg)

메시지가 포함된 초기화 에러.

err_msg must not be `NULL`.

PyStatus **PyStatus_NoMemory**(void)

메모리 할당 실패 (메모리 부족).

PyStatus **PyStatus_Exit**(int exitcode)

지정된 종료 코드로 파이썬을 종료합니다.

상태를 처리하는 함수:

int **PyStatus_Exception**(*PyStatus* status)

상태가 에러입니까? 아니면 종료입니까? 참이면, 예외를 처리해야 합니다; 예를 들어 `Py_ExitStatusException()` 을 호출하여.

int **PyStatus_IsError**(*PyStatus* status)

결과가 에러입니까?

int **PyStatus_IsExit**(*PyStatus* status)

결과가 종료입니까?

void **Py_ExitStatusException**(*PyStatus* status)

*status*가 종료이면 `exit(exitcode)` 를 호출합니다. *status*가 에러이면 에러 메시지를 인쇄하고 0이 아닌 종료 코드로 종료합니다. `PyStatus_Exception(status)` 가 0이 아닐 때만 호출해야 합니다.

참고

내부적으로, 파이썬은 `PyStatus.func`를 설정하는 데는 매크로를 사용하는 반면, `func`가 `NULL`로 설정된 상태를 만드는 데는 함수를 사용합니다.

예:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}
```

11.4.4 PyPreConfig

type **PyPreConfig**

Structure used to preinitialize Python.

사전 구성을 초기화하는 함수:

void **PyPreConfig_InitPythonConfig**(*PyPreConfig* *preconfig)

파이썬 구성으로 사전 구성을 초기화합니다.

void **PyPreConfig_InitIsolatedConfig**(*PyPreConfig* *preconfig)

격리된 구성으로 사전 구성을 초기화합니다.

구조체 필드:

int **allocator**

Name of the Python memory allocators:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): don' t change memory allocators (use defaults).
- `PYMEM_ALLOCATOR_DEFAULT` (`1`): *default memory allocators*.
- `PYMEM_ALLOCATOR_DEBUG` (`2`): *default memory allocators* with *debug hooks*.
- `PYMEM_ALLOCATOR_MALLOC` (`3`): use `malloc()` of the C library.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): force usage of `malloc()` with *debug hooks*.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): *Python pymalloc memory allocator*.
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): *Python pymalloc memory allocator* with *debug hooks*.
- `PYMEM_ALLOCATOR_MIMALLOC` (`6`): use `mimalloc`, a fast malloc replacement.
- `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` (`7`): use `mimalloc`, a fast malloc replacement with *debug hooks*.

`PYMEM_ALLOCATOR_PYMALLOC` and `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` are not supported if Python is `configured using --without-pymalloc`.

`PYMEM_ALLOCATOR_MIMALLOC` and `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` are not supported if Python is `configured using --without-mimalloc` or if the underlying atomic support isn' t available.

메모리 관리를 참조하십시오.

Default: `PYMEM_ALLOCATOR_NOT_SET`.

int **configure_locale**

Set the LC_CTYPE locale to the user preferred locale.

If equals to `0`, set *coerce_c_locale* and *coerce_c_locale_warn* members to `0`.

See the *locale encoding*.

Default: `1` in Python config, `0` in isolated config.

int **coerce_c_locale**

If equals to `2`, coerce the C locale.

If equals to `1`, read the LC_CTYPE locale to decide if it should be coerced.

See the *locale encoding*.

Default: `-1` in Python config, `0` in isolated config.

int **coerce_c_locale_warn**

0이 아니면, C 로케일이 강제될 때 경고가 발생합니다.

Default: `-1` in Python config, `0` in isolated config.

int **dev_mode**

Python Development Mode: see *PyConfig.dev_mode*.

Default: `-1` in Python mode, `0` in isolated mode.

int **isolated**

Isolated mode: see *PyConfig.isolated*.

Default: `0` in Python mode, `1` in isolated mode.

int **legacy_windows_fs_encoding**

If non-zero:

- Set *PyPreConfig.utf8_mode* to `0`,
- Set *PyConfig.filesystem_encoding* to `"mbcs"`,
- Set *PyConfig.filesystem_errors* to `"replace"`.

Initialized from the `PYTHONLEGACYWINDOWSFSENCODING` environment variable value.

윈도우에서만 사용 가능합니다. `#ifdef MS_WINDOWS` 매크로는 윈도우 특정 코드에 사용할 수 있습니다.

Default: `0`.

int **parse_argv**

0이 아니면, *Py_PreInitializeFromArgs()*와 *Py_PreInitializeFromBytesArgs()*는 일반 파이썬이 명령 줄 인자를 구문 분석하는 것과 같은 방식으로 `argv` 인자를 구문 분석합니다. 명령 줄 인자를 참조하십시오.

Default: `1` in Python config, `0` in isolated config.

int **use_environment**

Use environment variables? See *PyConfig.use_environment*.

Default: `1` in Python config and `0` in isolated config.

int **utf8_mode**

If non-zero, enable the Python UTF-8 Mode.

Set to `0` or `1` by the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

Also set to `1` if the `LC_CTYPE` locale is `C` or `POSIX`.

Default: `-1` in Python config and `0` in isolated config.

11.4.5 Preinitialize Python with PyPreConfig

The preinitialization of Python:

- Set the Python memory allocators (*PyPreConfig.allocator*)
- Configure the LC_CTYPE locale (*locale encoding*)
- Set the Python UTF-8 Mode (*PyPreConfig.utf8_mode*)

The current preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig`.

파이썬을 사전 초기화하는 함수:

PyStatus **Py_PreInitialize**(const *PyPreConfig* *preconfig)

preconfig 사전 구성에서 파이썬을 사전 초기화합니다.

preconfig must not be `NULL`.

PyStatus **Py_PreInitializeFromBytesArgs**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

preconfig 사전 구성에서 파이썬을 사전 초기화합니다.

Parse *argv* command line arguments (bytes strings) if *parse_argv* of *preconfig* is non-zero.

preconfig must not be `NULL`.

PyStatus **Py_PreInitializeFromArgs**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

preconfig 사전 구성에서 파이썬을 사전 초기화합니다.

Parse *argv* command line arguments (wide strings) if *parse_argv* of *preconfig* is non-zero.

preconfig must not be `NULL`.

호출자는 *PyStatus_Exception()*과 *Py_ExitStatusException()* 을 사용하여 예외(에러나 종료)를 처리해야 합니다.

For *Python Configuration* (*PyPreConfig_InitPythonConfig()*), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the `-X utf8` command line option enables the Python UTF-8 Mode.

PyMem_SetAllocator() 는 *Py_PreInitialize()* 이후에 *Py_InitializeFromConfig()* 이전에 호출하여 사용자 정의 메모리 할당자를 설치할 수 있습니다. *PyPreConfig.allocator*가 PYMEM_ALLOCATOR_NOT_SET으로 설정되면 *Py_PreInitialize()* 전에 호출할 수 있습니다.

Python memory allocation functions like *PyMem_RawMalloc()* must not be used before the Python preinitialization, whereas calling directly `malloc()` and `free()` is always safe. *Py_DecodeLocale()* must not be called before the Python preinitialization.

Example using the preinitialization to enable the Python UTF-8 Mode:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python speaks UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

11.4.6 PyConfig

type **PyConfig**

파이썬을 구성하기 위한 대부분의 파라미터를 포함하는 구조체.

When done, the *PyConfig_Clear()* function must be used to release the configuration memory.

구조체 메서드:

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

Initialize configuration with the *Python Configuration*.

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

Initialize configuration with the *Isolated Configuration*.

PyStatus **PyConfig_SetString**(*PyConfig* *config, wchar_t *const *config_str, const wchar_t *str)

와이드 문자열 *str*을 `*config_str`로 복사합니다.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesString**(*PyConfig* *config, wchar_t *const *config_str, const char *str)

Decode *str* using *Py_DecodeLocale()* and set the result into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv**(*PyConfig* *config, int argc, char *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of bytes strings. Decode bytes using *Py_DecodeLocale()*.

Preinitialize Python if needed.

PyStatus **PyConfig_SetWideStringList**(*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

와이드 문자열 리스트 *list*를 *length*와 *items*로 설정합니다.

Preinitialize Python if needed.

PyStatus **PyConfig_Read**(*PyConfig* *config)

모든 파이썬 구성을 읽습니다.

이미 초기화된 필드는 변경되지 않습니다.

Fields for *path configuration* are no longer calculated or modified when calling this function, as of Python 3.11.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once: *PyConfig.parse_argv* is set to `2` after arguments are parsed. Since Python arguments are stripped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Preinitialize Python if needed.

버전 3.10에서 변경: The *PyConfig.argv* arguments are now only parsed once, *PyConfig.parse_argv* is set to `2` after arguments are parsed, and arguments are only parsed if *PyConfig.parse_argv* equals `1`.

버전 3.11에서 변경: *PyConfig_Read()* no longer calculates all paths, and so fields listed under *Python Path Configuration* may no longer be updated until *Py_InitializeFromConfig()* is called.

void **PyConfig_Clear**(*PyConfig* *config)

구성 메모리를 해제합니다.

Most `PyConfig` methods *preinitialize Python* if needed. In that case, the Python preinitialization configuration (*PyPreConfig*) in based on the *PyConfig*. If configuration fields which are in common with *PyPreConfig* are tuned, they must be set before calling a *PyConfig* method:

- *PyConfig.dev_mode*
- *PyConfig.isolated*
- *PyConfig.parse_argv*
- *PyConfig.use_environment*

Moreover, if *PyConfig_SetArgv()* or *PyConfig_SetBytesArgv()* is used, this method must be called before other methods, since the preinitialization configuration depends on command line arguments (if *parse_argv* is non-zero).

이 메서드의 호출자는 `PyStatus_Exception()`과 `Py_ExitStatusException()`을 사용하여 예외(에러나 종료)를 처리해야 합니다.

구조체 필드:

PyWideStringList **argv**

Set `sys.argv` command line arguments based on *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string.

Set *parse_argv* to `1` to parse *argv* the same way the regular Python parses Python command line arguments and then to strip Python arguments from *argv*.

If *argv* is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

Default: `NULL`.

See also the *orig_argv* member.

int **safe_path**

If equals to zero, `Py_RunMain()` prepends a potentially unsafe path to `sys.path` at startup:

- If *`argv[0]`* is equal to `L"-m"` (`python -m module`), prepend the current working directory.
- If running a script (`python script.py`), prepend the script' s directory. If it' s a symbolic link, resolve symbolic links.
- Otherwise (`python -c code` and `python`), prepend an empty string, which means the current working directory.

Set to `1` by the `-P` command line option and the `PYTHONSAFEPATH` environment variable.

Default: `0` in Python config, `1` in isolated config.

Added in version 3.11.

wchar_t ***base_exec_prefix**

`sys.base_exec_prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.exec_prefix`*.

wchar_t ***base_executable**

Python base executable: `sys._base_executable`.

Set by the `__PYVENV_LAUNCHER__` environment variable.

Set from *`PyConfig.executable`* if `NULL`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.executable`*.

wchar_t ***base_prefix**

`sys.base_prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.prefix`*.

int **buffered_stdio**

If equals to `0` and *`configure_c_stdio`* is non-zero, disable buffering on the C streams stdout and stderr.

Set to `0` by the `-u` command line option and the `PYTHONUNBUFFERED` environment variable.

stdin은 항상 버퍼링 모드로 열립니다.

Default: `1`.

int **bytes_warning**

If equals to `1`, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`.

If equal or greater to `2`, raise a `BytesWarning` exception in these cases.

Incremented by the `-b` command line option.

Default: `0`.

int **warn_default_encoding**

If non-zero, emit a `EncodingWarning` warning when `io.TextIOWrapper` uses its default encoding. See io-encoding-warning for details.

Default: `0`.

Added in version 3.10.

int **code_debug_ranges**

If equals to `0`, disables the inclusion of the end line and column mappings in code objects. Also disables traceback printing carets to specific error locations.

Set to `0` by the `PYTHONNODEBUGRANGES` environment variable and by the `-X no_debug_ranges` command line option.

Default: `1`.

Added in version 3.11.

wchar_t ***check_hash_pycs_mode**

Control the validation behavior of hash-based `.pyc` files: value of the `--check-hash-based-pycs` command line option.

Valid values:

- `L"always"`: Hash the source file for invalidation regardless of value of the 'check_source' flag.
- `L"never"`: Assume that hash-based pycs always are valid.
- `L"default"`: The 'check_source' flag in hash-based pycs determines invalidation.

Default: `L"default"`.

See also **PEP 552** "Deterministic pycs".

int **configure_c_stdio**

If non-zero, configure C standard streams:

- On Windows, set the binary mode (`O_BINARY`) on stdin, stdout and stderr.
- If *`buffered_stdio`* equals zero, disable buffering of stdin, stdout and stderr streams.
- If *`interactive`* is non-zero, enable stream buffering on stdin and stdout (only stdout on Windows).

Default: `1` in Python config, `0` in isolated config.

int **dev_mode**

0이 아니면, 파이썬 개발 모드를 활성화합니다.

Set to `1` by the `-X dev` option and the `PYTHONDEVMODE` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int **dump_refs**

Dump Python references?

0이 아니면, 종료 시 여전히 활성 상태인 모든 객체를 덤프합니다.

Set to `1` by the `PYTHONDUMPREFS` environment variable.

Needs a special build of Python with the `Py_TRACE_REFS` macro defined: see the `configure --with-trace-refs option`.

Default: `0`.

wchar_t ***dump_refs_file**

Filename where to dump Python references.

Set by the `PYTHONDUMPREFSFILE` environment variable.

Default: `NULL`.

Added in version 3.11.

wchar_t ***exec_prefix**

The site-specific directory prefix where the platform-dependent Python files are installed: `sys.exec_prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.base_exec_prefix`*.

wchar_t ***executable**

The absolute path of the executable binary for the Python interpreter: `sys.executable`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.base_executable`*.

int **faulthandler**

Enable faulthandler?

0이 아니면, 시작 시 `faulthandler.enable()`을 호출합니다.

Set to `1` by `-X faulthandler` and the `PYTHONFAULTHANDLER` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

wchar_t ***filesystem_encoding**

Filesystem encoding: `sys.getfilesystemencoding()`.

On macOS, Android and VxWorks: use `"utf-8"` by default.

On Windows: use `"utf-8"` by default, or `"mbcs"` if *`legacy_windows_fs_encoding`* of *`PyPreConfig`* is non-zero.

Default encoding on other platforms:

- `"utf-8"` if *`PyPreConfig.utf8_mode`* is non-zero.
- `"ascii"` if Python detects that `nl_langinfo(CODESET)` announces the ASCII encoding, whereas the `mbstowcs()` function decodes from a different encoding (usually Latin1).
- `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string.
- Otherwise, use the *locale encoding*: `nl_langinfo(CODESET)` result.

At Python startup, the encoding name is normalized to the Python codec name. For example, `"ANSI_X3.4-1968"` is replaced with `"ascii"`.

See also the *`filesystem_errors`* member.

wchar_t ***filesystem_errors**

Filesystem error handler: `sys.getfilesystemencodeerrors()`.

On Windows: use `"surrogatepass"` by default, or `"replace"` if *`legacy_windows_fs_encoding`* of *`PyPreConfig`* is non-zero.

On other platforms: use `"surrogateescape"` by default.

Supported error handlers:

- `"strict"`
- `"surrogateescape"`
- `"surrogatepass"` (only supported with the UTF-8 encoding)

See also the *`filesystem_encoding`* member.

int **use_frozen_modules**

If non-zero, use frozen modules.

Set by the `PYTHON_FROZEN_MODULES` environment variable.

Default: `1` in a release build, or `0` in a debug build.

unsigned long **hash_seed**

int **use_hash_seed**

무작위 해시 함수 시드.

If *use_hash_seed* is zero, a seed is chosen randomly at Python startup, and *hash_seed* is ignored.

Set by the `PYTHONHASHSEED` environment variable.

Default *use_hash_seed* value: `-1` in Python mode, `0` in isolated mode.

wchar_t ***home**

Set the default Python "home" directory, that is, the location of the standard Python libraries (see `PYTHONHOME`).

Set by the `PYTHONHOME` environment variable.

Default: `NULL`.

Part of the *Python Path Configuration* input.

int **import_time**

If `1`, profile import time. If `2`, include additional output that indicates when an imported module has already been loaded.

Set by the `-X importtime` option and the `PYTHONPROFILEIMPORTTIME` environment variable.

Default: `0`.

버전 3.14에서 변경: Added support for `import_time = 2`

int **inspect**

스크립트나 명령을 실행한 후 대화식 모드로 들어갑니다.

If greater than `0`, enable inspect: when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Incremented by the `-i` command line option. Set to `1` if the `PYTHONINSPECT` environment variable is non-empty.

Default: `0`.

int **install_signal_handlers**

Install Python signal handlers?

Default: `1` in Python mode, `0` in isolated mode.

int **interactive**

If greater than `0`, enable the interactive mode (REPL).

Incremented by the `-i` command line option.

Default: `0`.

int **int_max_str_digits**

Configures the integer string conversion length limitation. An initial value of `-1` means the value will be taken from the command line or environment or otherwise default to 4300 (`sys.int_info.default_max_str_digits`). A value of `0` disables the limitation. Values greater than zero but less

than 640 (`sys.int_info.str_digits_check_threshold`) are unsupported and will produce an error.

Configured by the `-X int_max_str_digits` command line flag or the `PYTHONINTMAXSTRDIGITS` environment variable.

Default: `-1` in Python mode. 4300 (`sys.int_info.default_max_str_digits`) in isolated mode.

Added in version 3.12.

int **cpu_count**

If the value of *cpu_count* is not `-1` then it will override the return values of `os.cpu_count()`, `os.process_cpu_count()`, and `multiprocessing.cpu_count()`.

Configured by the `-X cpu_count=n|default` command line flag or the `PYTHON_CPU_COUNT` environment variable.

Default: `-1`.

Added in version 3.13.

int **isolated**

If greater than `0`, enable isolated mode:

- Set *safe_path* to `1`: don' t prepend a potentially unsafe path to `sys.path` at Python startup, such as the current directory, the script' s directory or an empty string.
- Set *use_environment* to `0`: ignore `PYTHON` environment variables.
- Set *user_site_directory* to `0`: don' t add the user site directory to `sys.path`.
- 파이썬 REPL은 대화식 프롬프트에서 `readline`을 임포트 하지도 기본 readline 구성을 활성화하지도 않습니다.

Set to `1` by the `-I` command line option.

Default: `0` in Python mode, `1` in isolated mode.

See also the *Isolated Configuration* and *PyPreConfig.isolated*.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

윈도우에서만 사용 가능합니다. `#ifdef MS_WINDOWS` 매크로는 윈도우 특정 코드에 사용할 수 있습니다.

Default: `0`.

See also the **PEP 528** (Change Windows console encoding to UTF-8).

int **malloc_stats**

0이 아니면, 종료 시 파이썬 *pymalloc* 메모리 할당자에 대한 통계를 덤프합니다.

Set to `1` by the `PYTHONMALLOCSTATS` environment variable.

The option is ignored if Python is `configured using the --without-pymalloc option`.

Default: `0`.

wchar_t ***platlibdir**

Platform library directory name: `sys.platlibdir`.

Set by the `PYTHONPLATLIBDIR` environment variable.

Default: value of the `PLATLIBDIR` macro which is set by the `configure --with-platlibdir option` (default: `"lib"`, or `"DLLs"` on Windows).

Part of the *Python Path Configuration* input.

Added in version 3.9.

버전 3.11에서 변경: This macro is now used on Windows to locate the standard library extension modules, typically under `DLLs`. However, for compatibility, note that this value is ignored for any non-standard layouts, including in-tree builds and virtual environments.

wchar_t ***pythonpath_env**

Module search paths (`sys.path`) as a string separated by `DELIM` (`os.pathsep`).

Set by the `PYTHONPATH` environment variable.

Default: `NULL`.

Part of the *Python Path Configuration* input.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

Module search paths: `sys.path`.

If *module_search_paths_set* is equal to `0`, *Py_InitializeFromConfig()* will replace *module_search_paths* and sets *module_search_paths_set* to `1`.

Default: empty list (`module_search_paths`) and `0` (`module_search_paths_set`).

Part of the *Python Path Configuration* output.

int **optimization_level**

컴파일 최적화 수준:

- `0`: Peephole optimizer, set `__debug__` to `True`.
- `1`: Level 0, remove assertions, set `__debug__` to `False`.
- `2`: Level 1, strip docstrings.

Incremented by the `-O` command line option. Set to the `PYTHONOPTIMIZE` environment variable value.

Default: `0`.

PyWideStringList **orig_argv**

The list of the original command line arguments passed to the Python executable: `sys.orig_argv`.

If *orig_argv* list is empty and *argv* is not a list only containing an empty string, *PyConfig_Read()* copies *argv* into *orig_argv* before modifying *argv* (if *parse_argv* is non-zero).

See also the *argv* member and the *Py_GetArgcArgv()* function.

Default: empty list.

Added in version 3.10.

int **parse_argv**

Parse command line arguments?

If equals to `1`, parse *argv* the same way the regular Python parses command line arguments, and strip Python arguments from *argv*.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once: *PyConfig.parse_argv* is set to `2` after arguments are parsed. Since Python arguments are stripped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Default: `1` in Python mode, `0` in isolated mode.

버전 3.10에서 변경: The *PyConfig.argv* arguments are now only parsed if *PyConfig.parse_argv* equals to `1`.

int **parser_debug**

Parser debug mode. If greater than `0`, turn on parser debugging output (for expert only, depending on compilation options).

Incremented by the `-d` command line option. Set to the `PYTHONDEBUG` environment variable value.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default: `0`.

int **pathconfig_warnings**

If non-zero, calculation of path configuration is allowed to log warnings into `stderr`. If equals to `0`, suppress these warnings.

Default: `1` in Python mode, `0` in isolated mode.

Part of the *Python Path Configuration* input.

버전 3.11에서 변경: Now also applies on Windows.

wchar_t ***prefix**

The site-specific directory prefix where the platform independent Python files are installed: `sys.prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.base_prefix*.

wchar_t ***program_name**

Program name used to initialize *executable* and in early error messages during Python initialization.

- On macOS, use `PYTHONEXECUTABLE` environment variable if set.
- If the `WITH_NEXT_FRAMEWORK` macro is defined, use `__PYVENV_LAUNCHER__` environment variable if set.
- Use `argv[0]` of *argv* if available and non-empty.
- Otherwise, use `L"python"` on Windows, or `L"python3"` on other platforms.

Default: `NULL`.

Part of the *Python Path Configuration* input.

wchar_t ***pycache_prefix**

Directory where cached `.pyc` files are written: `sys.pycache_prefix`.

Set by the `-X pycache_prefix=PATH` command line option and the `PYTHONPYCACHEPREFIX` environment variable. The command-line option takes precedence.

`NULL`이면, `sys.pycache_prefix`는 `None`으로 설정됩니다.

Default: `NULL`.

int **quiet**

Quiet mode. If greater than `0`, don' t display the copyright and version at Python startup in interactive mode.

Incremented by the `-q` command line option.

Default: `0`.

wchar_t ***run_command**

Value of the `-c` command line option.

Used by *Py_RunMain()*.

Default: `NULL`.

wchar_t ***run_filename**

Filename passed on the command line: trailing command line argument without `-c` or `-m`. It is used by the *Py_RunMain()* function.

For example, it is set to `script.py` by the `python3 script.py arg` command line.

See also the *PyConfig.skip_source_first_line* option.

Default: `NULL`.

wchar_t ***run_module**

Value of the `-m` command line option.

Used by *Py_RunMain()*.

Default: `NULL`.

wchar_t ***run_presite**

`package.module` path to module that should be imported before `site.py` is run.

Set by the `-X presite=package.module` command-line option and the `PYTHON_PRESITE` environment variable. The command-line option takes precedence.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default: `NULL`.

int **show_ref_count**

Show total reference count at exit (excluding *immortal* objects)?

Set to `1` by `-X showrefcount` command line option.

Needs a debug build of Python (the `Py_REF_DEBUG` macro must be defined).

Default: `0`.

int **site_import**

시작할 때 `site` 모듈을 임포트 합니까?

If equal to zero, disable the import of the module site and the site-dependent manipulations of `sys.path` that it entails.

Also disable these manipulations if the `site` module is explicitly imported later (call `site.main()` if you want them to be triggered).

Set to `0` by the `-S` command line option.

`sys.flags.no_site` is set to the inverted value of *site_import*.

Default: `1`.

int **skip_source_first_line**

If non-zero, skip the first line of the *PyConfig.run_filename* source.

It allows the usage of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

Set to `1` by the `-x` command line option.

Default: `0`.

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr` (but `sys.stderr` always uses `"backslashreplace"` error handler).

Use the `PYTHONIOENCODING` environment variable if it is non-empty.

Default encoding:

- `"UTF-8"` if *PyPreConfig.utf8_mode* is non-zero.

- Otherwise, use the *locale encoding*.

Default error handler:

- On Windows: use `"surrogateescape"`.
- `"surrogateescape"` if *PyPreConfig.utf8_mode* is non-zero, or if the LC_CTYPE locale is "C" or "POSIX".
- `"strict"` otherwise.

See also *PyConfig.legacy_windows_stdio*.

int **tracemalloc**

Enable tracemalloc?

0이 아니면, 시작 시 `tracemalloc.start()`를 호출합니다.

Set by `-X tracemalloc=N` command line option and by the `PYTHONTRACEMALLOC` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int **perf_profiling**

Enable the Linux `perf` profiler support?

If equals to `1`, enable support for the Linux `perf` profiler.

If equals to `2`, enable support for the Linux `perf` profiler with DWARF JIT support.

Set to `1` by `-X perf` command-line option and the `PYTHONPERFSUPPORT` environment variable.

Set to `2` by the `-X perf_jit` command-line option and the `PYTHON_PERF_JIT_SUPPORT` environment variable.

Default: `-1`.

더 보기

See perf_profiling for more information.

Added in version 3.12.

wchar_t ***stdlib_dir**

Directory of the Python standard library.

Default: `NULL`.

Added in version 3.11.

int **use_environment**

Use environment variables?

If equals to zero, ignore the environment variables.

Set to `0` by the `-E` environment variable.

Default: `1` in Python config and `0` in isolated config.

int **use_system_logger**

If non-zero, `stdout` and `stderr` will be redirected to the system log.

Only available on macOS 10.12 and later, and on iOS.

Default: `0` (don' t use the system log) on macOS; `1` on iOS (use the system log).

Added in version 3.14.

int **user_site_directory**

If non-zero, add the user site directory to `sys.path`.

Set to `0` by the `-s` and `-I` command line options.

Set to `0` by the `PYTHONNOUSERSITE` environment variable.

Default: `1` in Python mode, `0` in isolated mode.

int **verbose**

Verbose mode. If greater than `0`, print a message each time a module is imported, showing the place (filename or built-in module) from which it is loaded.

If greater than or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Incremented by the `-v` command line option.

Set by the `PYTHONVERBOSE` environment variable value.

Default: `0`.

PyWideStringList **warnoptions**

Options of the `warnings` module to build warnings filters, lowest to highest priority: `sys.warnoptions`.

`warnings` 모듈은 `sys.warnoptions`를 역순으로 추가합니다: 마지막 *PyConfig.warnoptions* 항목은 가장 먼저 검사되는 `warnings.filters`의 첫 번째 항목이 됩니다 (가장 높은 우선순위).

The `-W` command line options adds its value to *warnoptions*, it can be used multiple times.

The `PYTHONWARNINGS` environment variable can also be used to add warning options. Multiple options can be specified, separated by commas (`,`).

Default: empty list.

int **write_bytecode**

If equal to `0`, Python won' t try to write `.pyc` files on the import of source modules.

Set to `0` by the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable.

`sys.dont_write_bytecode`는 *write_bytecode*의 반전된 값으로 초기화됩니다.

Default: `1`.

PyWideStringList **xoptions**

Values of the `-X` command line options: `sys._xoptions`.

Default: empty list.

int **_pystats**

If non-zero, write performance statistics at Python exit.

Need a special build with the `Py_STATS` macro: see `--enable-pystats`.

Default: `0`.

If *parse_argv* is non-zero, *argv* arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from *argv*.

The *xoptions* options are parsed to set other options: see the `-X` command line option.

버전 3.9에서 변경: `show_alloc_count` 필드가 제거되었습니다.

11.4.7 PyConfig를 사용한 초기화

Initializing the interpreter from a populated configuration struct is handled by calling *Py_InitializeFromConfig()*.

호출자는 *PyStatus_Exception()*과 *Py_ExitStatusException()* 을 사용하여 예외(에러나 종료)를 처리해야 합니다.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

The current configuration (`PyConfig` type) is stored in `PyInterpreterState.config`.

프로그램 이름을 설정하는 예:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters. Note that since 3.11, many parameters are not calculated until initialization, and so values cannot be read from the configuration structure. Any values set before initialize is called will be left unchanged by initialization:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

11.4.8 격리된 구성

*PyPreConfig_InitIsolatedConfig()*와 *PyConfig_InitIsolatedConfig()* 함수는 시스템에서 파이썬을 격리하는 구성을 만듭니다. 예를 들어, 파이썬을 응용 프로그램에 내장하기 위해.

This configuration ignores global configuration variables, environment variables, command line arguments (*PyConfig.argv* is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the LC_CTYPE locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure *PyConfig.home* is specified to avoid computing the default path configuration.

11.4.9 파이썬 구성

*PyPreConfig_InitPythonConfig()*와 *PyConfig_InitPythonConfig()* 함수는 일반 파이썬처럼 동작하는 사용자 정의된 파이썬을 빌드하기 위한 구성을 만듭니다.

환경 변수와 명령 줄 인자는 파이썬을 구성하는 데 사용되는 반면, 전역 구성 변수는 무시됩니다.

This function enables C locale coercion (**PEP 538**) and Python UTF-8 Mode (**PEP 540**) depending on the LC_CTYPE locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

11.4.10 Python Path Configuration

*PyConfig*에는 경로 구성을 위한 여러 필드가 포함되어 있습니다:

- 경로 구성 입력:
  - *PyConfig.home*
  - *PyConfig.platlibdir*
  - *PyConfig.pathconfig_warnings*
  - *PyConfig.program_name*
  - *PyConfig.pythonpath_env*
  - 현재 작업 디렉터리: 절대 경로를 얻기 위해
  - (*PyConfig.program_name*에서) 프로그램 전체 경로를 얻기 위한 `PATH` 환경 변수
  - `__PYVENV_LAUNCHER__` 환경 변수
  - (윈도우 전용) HKEY_CURRENT_USER 와 HKEY_LOCAL_MACHINE 의 "SoftwarePythonPythonCoreX.YPythonPath" 아래에 있는 레지스트리의 응용 프로그램 경로 (여기서 X.Y는 파이썬 버전입니다).
- 경로 구성 출력 필드:
  - *PyConfig.base_exec_prefix*
  - *PyConfig.base_executable*
  - *PyConfig.base_prefix*
  - *PyConfig.exec_prefix*
  - *PyConfig.executable*
  - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
  - *PyConfig.prefix*

If at least one "output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to `0`, *module_search_paths* is overridden and *module_search_paths_set* is set to `1`.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. `module_search_paths` is considered as set if `module_search_paths_set` is set to `1`. In this case, `module_search_paths` will be used without modification.

Set *pathconfig_warnings* to `0` to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

*base_prefix*나 *base_exec_prefix* 필드가 설정되지 않으면, 각각 *prefix*와 *exec_prefix*의 값을 상속합니다.

*Py_RunMain()*과 *Py_Main()*은 `sys.path`를 수정합니다:

- *run_filename*이 설정되고 `__main__.py` 스크립트를 포함하는 디렉터리이면, *run_filename*을 `sys.path` 앞에 추가합니다.
- *isolated*가 0이면:
  - *run_module*이 설정되면, 현재 디렉터리를 `sys.path` 앞에 추가합니다. 현재 디렉터리를 읽을 수 없으면 아무것도 하지 않습니다.
  - *run_filename*이 설정되면, 파일명의 디렉터리를 `sys.path` 앞에 추가합니다.

- 그렇지 않으면, 빈 문자열을 `sys.path` 앞에 추가합니다.

*site_import*가 0이 아니면, `site` 모듈이 `sys.path`를 수정할 수 있습니다. *user_site_directory*가 0이 아니고 사용자의 site-package 디렉터리가 존재하면, `site` 모듈은 사용자의 site-package 디렉터리를 `sys.path`에 추가합니다.

다음과 같은 구성 파일이 경로 구성에 사용됩니다:

- `pyvenv.cfg`
- `._pth` file (ex: `python._pth`)
- `pybuilddir.txt` (유닉스 전용)

If a `._pth` file is present:

- Set *isolated* to `1`.
- Set *use_environment* to `0`.
- Set *site_import* to `0`.
- Set *safe_path* to `1`.

If *home* is not set and a `pyvenv.cfg` file is present in the same directory as *executable*, or its parent, *prefix* and *exec_prefix* are set that location. When this happens, *base_prefix* and *base_exec_prefix* still keep their value, pointing to the base installation. See sys-path-init-virtual-environments for more information.

The `__PYVENV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*.

버전 3.14에서 변경: *prefix*, and *exec_prefix*, are now set to the `pyvenv.cfg` directory. This was previously done by `site`, therefore affected by `-S`.

11.5 Py_GetArgcArgv()

void **Py_GetArgcArgv**(int *argc, wchar_t ***argv)

파이썬이 수정하기 전의, 원래 명령 줄 인자를 가져옵니다.

See also *PyConfig.orig_argv* member.

11.6 Delaying main module execution

In some embedding use cases, it may be desirable to separate interpreter initialization from the execution of the main module.

This separation can be achieved by setting `PyConfig.run_command` to the empty string during initialization (to prevent the interpreter from dropping into the interactive prompt), and then subsequently executing the desired main module code using `__main__.__dict__` as the global namespace.

CHAPTER 12

메모리 관리

12.1 개요

파이썬의 메모리 관리에는 모든 파이썬 객체와 데이터 구조를 포함하는 비공개 힙(private heap)을 수반합니다. 이 비공개 힙의 관리는 파이썬 메모리 관리자에 의해 내부적으로 이루어집니다. 파이썬 메모리 관리자는 공유, 세그먼트 화, 사전 할당 또는 캐싱과 같은 동적 스토리지 관리의 다양한 측면을 처리하는 서로 다른 구성 요소를 가지고 있습니다.

가장 낮은 수준에서, 원시 메모리 할당자는 운영 체제의 메모리 관리자와 상호 작용하여 비공개 힙에 모든 파이썬 관련 데이터를 저장하기에 충분한 공간이 있는지 확인합니다. 원시 메모리 할당자 위에, 여러 개의 객체별 할당자가 같은 힙에서 작동하며 각 객체 형의 특성에 맞는 고유한 메모리 관리 정책을 구현합니다. 예를 들어, 정수는 다른 스토리지 요구 사항과 속도/공간 절충을 의미하므로, 정수 객체는 힙 내에서 문자열, 튜플 또는 딕셔너리와는 다르게 관리됩니다. 따라서 파이썬 메모리 관리자는 일부 작업을 객체별 할당자에게 위임하지만, 후자가 비공개 힙의 경계 내에서 작동하도록 합니다.

파이썬 힙의 관리는 인터프리터 자체에 의해 수행되며, 사용자는 힙 내부의 메모리 블록에 대한 객체 포인터를 규칙적으로 조작하더라도, 사용자가 제어할 수 없다는 것을 이해하는 것이 중요합니다. 파이썬 객체와 기타 내부 버퍼를 위한 힙 공간 할당은 이 설명서에 나열된 파이썬/C API 함수를 통해 파이썬 메모리 관리자의 요청에 따라 수행됩니다.

메모리 손상을 피하고자, 확장 작성자는 C 라이브러리에서 내보낸 함수를 파이썬 객체에 대해 실행하지 않아야 합니다: `malloc()`, `calloc()`, `realloc()` 및 `free()`. 그렇게 한다면, 서로 다른 알고리즘을 구현하고 다른 힘에 작동하기 때문에, C 할당자와 파이썬 메모리 관리자 간에 혼합 호출이 발생하여 치명적인 결과를 초래합니다. 그러나, 다음 예제와 같이 개별 목적으로 C 라이브러리 할당자를 사용하여 메모리 블록을 안전하게 할당하고 해제할 수 있습니다:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* I/O 용 */

if (buf == NULL)
    return PyErr_NoMemory();
...buf 를 사용하는 I/O 연산을 수행합니다...
res = PyBytes_FromString(buf);
free(buf); /* malloc 으로 할당했습니다 */
return res;
```

이 예에서, I/O 버퍼에 대한 메모리 요청은 C 라이브러리 할당자에 의해 처리됩니다. 파이썬 메모리 관리자는 결과로 반환되는 바이트열 객체의 할당에만 관여합니다.

그러나 대부분의 경우, 파이썬 힙에서 메모리를 할당하는 것이 좋습니다. 파이썬 힙은 파이썬 메모리 관리자가 제어하기 때문입니다. 예를 들어, 인터프리터가 C로 작성된 새로운 객체 형으로 확장될 때 필요합니다. 파이썬 힙을 사용하는 또 다른 이유는 확장 모듈의 메모리 요구에 대해 파이썬 메모리 관리자에게 알리고자 하는 것입니다. 요청된 메모리가 내부적이고 매우 특정한 목적으로만 사용될 때도, 모든 메모리 요청을 파이썬 메모리 관리자에 위임하면 인터프리터가 전체 메모리 요구량에 대한 더 정확한 이미지를 갖게 됩니다. 결과적으로, 특정 상황에서, 파이썬 메모리 관리자는 가비지 수집, 메모리 압축 또는 기타 예방 절차와 같은 적절한 작업을 트리거 하거나 그러지 않을 수 있습니다. 앞의 예에서와같이 C 라이브러리 할당자를 사용하면, I/O 버퍼에 할당된 메모리가 파이썬 메모리 관리자를 완전히 우회하게 됨에 유의하십시오.

더 보기

`PYTHONMALLOC` 환경 변수를 사용하여 파이썬에서 사용하는 메모리 할당자를 구성할 수 있습니다.

`PYTHONMALLOCSTATS` 환경 변수는 새로운 pymalloc 객체 아레나(arena)가 만들어질 때마다 그리고 종료 시 *pymalloc 메모리 할당자*의 통계를 인쇄하는 데 사용될 수 있습니다.

12.2 Allocator Domains

All allocating functions belong to one of three different “domains” (see also *PyMemAllocatorDomain*). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at *here*. The APIs used to allocate and free a block of memory must be from the same domain. For example, *PyMem_Free()* must be used to free memory allocated using *PyMem_Malloc()*.

The three allocation domains are:

- Raw domain: intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without an *attached thread state*. The memory is requested directly from the system. See *Raw Memory Interface*.
- “Mem” domain: intended for allocating memory for Python buffers and general-purpose memory buffers where the allocation must be performed with an *attached thread state*. The memory is taken from the Python private heap. See *Memory Interface*.
- Object domain: intended for allocating memory for Python objects. The memory is taken from the Python private heap. See *Object allocators*.

참고

The *free-threaded* build requires that only Python objects are allocated using the “object” domain and that all Python objects are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

For example, buffers (non-Python objects) should be allocated using *PyMem_Malloc()*, *PyMem_RawMalloc()*, or `malloc()`, but not *PyObject_Malloc()*.

See Memory Allocation APIs.

12.3 원시 메모리 인터페이스

The following function sets are wrappers to the system allocator. These functions are thread-safe, so a *thread state* does not need to be *attached*.

기본 원시 메모리 할당자는 다음 함수를 사용합니다: `malloc()`, `calloc()`, `realloc()` 및 `free()`; 0바이트를 요청할 때 `malloc(1)`(또는 `calloc(1, 1)`)을 호출합니다.

Added in version 3.4.

void *__PyMem_RawMalloc__(size_t n)

Part of the 안정 ABI 버전 *3.13* 이후로. *n* 바이트를 할당하고 할당된 메모리를 가리키는 `void*` 형의 포인터를 반환하거나, 요청이 실패하면 `NULL`을 반환합니다.

0바이트를 요청하면 가능하면 `PyMem_RawMalloc(1)` 이 대신 호출된 것처럼 가능하면 고유한 `NULL` 이 아닌 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

void *__PyMem_RawCalloc__(size_t nelem, size_t elsize)

Part of the 안정 ABI 버전 *3.13* 이후로. 크기가 각각 *elsize* 바이트인 *nelem* 개의 요소를 할당하고 할당된 메모리를 가리키는 `void*` 형의 포인터를 반환하거나, 요청이 실패하면 `NULL`을 반환합니다. 메모리는 0으로 초기화됩니다.

0개의 요소나 0바이트 크기의 요소를 요청하면 `PyMem_RawCalloc(1, 1)` 이 대신 호출된 것처럼 가능하면 고유한 `NULL`이 아닌 포인터를 반환합니다.

Added in version 3.5.

void *__PyMem_RawRealloc__(void *p, size_t n)

Part of the 안정 ABI 버전 *3.13* 이후로. *p*가 가리키는 메모리 블록의 크기를 *n* 바이트로 조정합니다. 내용은 이전과 새로운 크기의 최솟값 내에서는 변경되지 않습니다.

*p*가 `NULL`이면, 호출은 `PyMem_RawMalloc(n)` 과 동등합니다; *n*이 0과 같으면, 메모리 블록의 크기는 조정되지만 해제되지는 않고, 반환된 포인터는 `NULL`이 아닙니다.

*p*가 `NULL`이 아닌 한, *PyMem_RawMalloc()*, *PyMem_RawRealloc()* 또는 *PyMem_RawCalloc()*에 대한 이전 호출에 의해 반환된 것이어야 합니다.

요청이 실패하면, *PyMem_RawRealloc()*은 `NULL`을 반환하고 *p*는 이전 메모리 영역에 대한 유효한 포인터로 유지됩니다.

void __PyMem_RawFree__(void *p)

Part of the 안정 ABI 버전 *3.13* 이후로. *p*가 가리키는 메모리 블록을 해제합니다. *p*는 *PyMem_RawMalloc()*, *PyMem_RawRealloc()* 또는 *PyMem_RawCalloc()*에 대한 이전 호출로 반환된 것이어야 합니다. 그렇지 않거나 `PyMem_RawFree(p)` 가 앞서 호출되었으면, 정의되지 않은 동작이 일어납니다.

*p*가 `NULL`이면, 아무 작업도 수행되지 않습니다.

12.4 메모리 인터페이스

ANSI C 표준에 따라 모델링 되었지만 0바이트를 요청할 때의 동작을 지정한 다음 함수 집합은 파이썬 힙에서 메모리를 할당하고 해제하는 데 사용할 수 있습니다.

기본 메모리 할당자는 *pymalloc* 메모리 할당자를 사용합니다.

경고

There must be an *attached thread state* when using these functions.

버전 3.6에서 변경: 기본 할당자는 이제 시스템 `malloc()` 대신 pymalloc 입니다.

void *__PyMem_Malloc__(size_t n)

Part of the 안정 ABI. *n* 바이트를 할당하고 할당된 메모리를 가리키는 `void*` 형의 포인터를 반환하거나, 요청이 실패하면 `NULL`을 반환합니다.

0바이트를 요청하면 `PyMem_Malloc(1)` 이 대신 호출된 것처럼 가능하면 고유한 `NULL`이 아닌 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

void *__PyMem_Calloc__(size_t nelem, size_t elsize)

Part of the 안정 ABI 버전 *3.7* 이후로. 크기가 각각 *elsize* 바이트인 *nelem* 개의 요소를 할당하고 할당된 메모리를 가리키는 `void*` 형의 포인터를 반환하거나, 요청이 실패하면 `NULL`을 반환합니다. 메모리는 0으로 초기화됩니다.

0개의 요소나 0바이트 크기의 요소를 요청하면 `PyMem_Calloc(1, 1)`이 대신 호출된 것처럼 가능하면 고유한 `NULL`이 아닌 포인터를 반환합니다.

Added in version 3.5.

void ***PyMem_Realloc**(void *p, size_t n)

Part of the 안정 ABI. *p*가 가리키는 메모리 블록의 크기를 *n* 바이트로 조정합니다. 내용은 이전과 새로운 크기의 최솟값 내에서는 변경되지 않습니다.

*p*가 `NULL`이면, 호출은 `PyMem_Malloc(n)`과 동등합니다; 그렇지 않고 *n*이 0과 같으면, 메모리 블록의 크기는 조정되지만 해제되지는 않으며, 반환된 포인터는 `NULL`이 아닙니다.

*p*가 `NULL`이 아닌 한, *PyMem_Malloc()*, *PyMem_Realloc()* 또는 *PyMem_Calloc()*에 대한 이전 호출이 반환한 것이어야 합니다.

요청이 실패하면, *PyMem_Realloc()*은 `NULL`을 반환하고 *p*는 이전 메모리 영역에 대한 유효한 포인터로 유지됩니다.

void **PyMem_Free**(void *p)

Part of the 안정 ABI. *p*가 가리키는 메모리 블록을 해제합니다. *p*는 *PyMem_Malloc()*, *PyMem_Realloc()* 또는 *PyMem_Calloc()*에 대한 이전 호출이 반환한 것이어야 합니다. 그렇지 않거나 `PyMem_Free(p)`가 앞서 호출되었으면 정의되지 않은 동작이 일어납니다.

*p*가 `NULL`이면, 아무 작업도 수행되지 않습니다.

편의를 위해 다음과 같은 형 지향 매크로가 제공됩니다. *TYPE*이 모든 C 형을 나타냄에 유의하십시오.

PyMem_New(TYPE, n)

*PyMem_Malloc()*과 같지만, `(n * sizeof(TYPE))` 바이트의 메모리를 할당합니다. `TYPE*`로 캐스트된 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

PyMem_Resize(p, TYPE, n)

*PyMem_Realloc()*과 같지만, 메모리 블록의 크기는 `(n * sizeof(TYPE))` 바이트로 조정됩니다. `TYPE*`로 캐스트 된 포인터를 반환합니다. 반환한 후에, *p*는 새로운 메모리 영역에 대한 포인터이거나, 실패하면 `NULL`이 됩니다.

이것은 C 전처리기 매크로입니다; *p*는 항상 다시 대입됩니다. 에러를 처리할 때 메모리 손실을 피하려면 *p*의 원래 값을 보관하십시오.

void **PyMem_Del**(void *p)

*PyMem_Free()*와 같습니다.

또한, 위에 나열된 C API 함수를 사용하지 않고, 파이썬 메모리 할당자를 직접 호출하기 위해 다음 매크로 집합이 제공됩니다. 그러나, 이들을 사용하면 파이썬 버전을 가로지르는 바이너리 호환성이 유지되지 않아서 확장 모듈에서는 폐지되었습니다.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

12.5 객체 할당자

ANSI C 표준에 따라 모델링 되었지만 0바이트를 요청할 때의 동작을 지정한 다음 함수 집합은 파이썬 힙에서 메모리를 할당하고 해제하는 데 사용할 수 있습니다.

참고

There is no guarantee that the memory returned by these allocators can be successfully cast to a Python object when intercepting the allocating functions in this domain by the methods described in the *Customize Memory Allocators* section.

기본 객체 할당자는 *pymalloc* 메모리 할당자를 사용합니다.

경고

There must be an *attached thread state* when using these functions.

void ***PyObject_Malloc**(size_t n)

Part of the 안정 ABI. *n* 바이트를 할당하고 할당된 메모리를 가리키는 `void*` 형의 포인터를 반환하거나, 요청이 실패하면 `NULL`을 반환합니다.

0바이트를 요청하면 `PyObject_Malloc(1)` 이 대신 호출된 것처럼 가능하면 고유한 `NULL`이 아닌 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

void ***PyObject_Calloc**(size_t nelem, size_t elsize)

Part of the 안정 ABI 버전 *3.7* 이후로. 크기가 각각 *elsize* 바이트인 *nelem* 개의 요소를 할당하고 할당된 메모리를 가리키는 `void*` 형의 포인터를 반환하거나, 요청이 실패하면 `NULL`을 반환합니다. 메모리는 0으로 초기화됩니다.

0개의 요소나 0바이트 크기의 요소를 요청하면 `PyObject_Calloc(1, 1)` 이 대신 호출된 것처럼 가능하면 고유한 `NULL`이 아닌 포인터를 반환합니다.

Added in version 3.5.

void ***PyObject_Realloc**(void *p, size_t n)

Part of the 안정 ABI. *p*가 가리키는 메모리 블록의 크기를 *n* 바이트로 조정합니다. 내용은 이전과 새로운 크기의 최솟값 내에서는 변경되지 않습니다.

*p*가 `NULL`이면, 호출은 `PyObject_Malloc(n)` 과 동등합니다; 그렇지 않고 *n*이 0과 같으면, 메모리 블록의 크기는 조정되지만 해제되지 않고, 반환된 포인터는 `NULL`이 아닙니다.

*p*가 `NULL`이 아닌 한, *PyObject_Malloc()*, *PyObject_Realloc()* 또는 *PyObject_Calloc()*에 대한 이전 호출에 의해 반환된 것이어야 합니다.

요청이 실패하면, *PyObject_Realloc()*은 `NULL`을 반환하고 *p*는 이전 메모리 영역에 대한 유효한 포인터로 유지됩니다.

void **PyObject_Free**(void *p)

Part of the 안정 ABI. *p*가 가리키는 메모리 블록을 해제합니다. 이 블록은 *PyObject_Malloc()*, *PyObject_Realloc()* 또는 *PyObject_Calloc()*에 대한 이전 호출에 의해 반환된 것이어야 합니다. 그렇지 않거나 `PyObject_Free(p)` 가 이전에 호출되었으면 정의되지 않은 동작이 일어납니다.

*p*가 `NULL`이면, 아무 작업도 수행되지 않습니다.

Do not call this directly to free an object' s memory; call the type' s *tp_free* slot instead.

Do not use this for memory allocated by *PyObject_GC_New* or *PyObject_GC_NewVar*; use *PyObject_GC_Del()* instead.

더 보기

- *PyObject_GC_Del()* is the equivalent of this function for memory allocated by types that support garbage collection.
- *PyObject_Malloc()*

- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_New*
- *PyObject_NewVar*
- *PyType_GenericAlloc()*
- *tp_free*

12.6 기본 메모리 할당자

기본 메모리 할당자:

구성	이름	PyMem_RawMallo	PyMem_Malloc	PyOb-ject_Malloc
릴리스 빌드	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
디버그 빌드	`"pymalloc_debug`	`malloc` + 디버그	`pymalloc` + 디버그	`pymalloc` + 디버그
pymalloc 없는 배포 빌드	`"malloc"`	`malloc`	`malloc`	`malloc`
pymalloc 없는 디버그 빌드	`"malloc_debug"`	`malloc` + 디버그	`malloc` + 디버그	`malloc` + 디버그

범례:

- 이름: `PYTHONMALLOC` 환경 변수의 값.
- `malloc`: 표준 C 라이브러리의 시스템 할당자, C 함수: `malloc()`, `calloc()`, `realloc()` 및 `free()`.
- `pymalloc`: *pymalloc* 메모리 할당자.
- `mimalloc`: *mimalloc memory allocator*. The pymalloc allocator will be used if mimalloc support isn' t available.
- "+ 디버그": 파이썬 메모리 할당기의 디버그 훅 포함.
- "Debug build": Python build in debug mode.

12.7 메모리 할당자 사용자 정의

Added in version 3.4.

type **PyMemAllocatorEx**

메모리 블록 할당자를 기술하는 데 사용되는 구조체. 구조체에는 다음과 같은 필드가 있습니다:

필드	의미
`void *ctx`	첫 번째 인자로 전달된 사용자 컨텍스트
`void* malloc(void *ctx, size_t size)`	메모리 블록을 할당합니다
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	0으로 초기화된 메모리 블록을 할당합니다
`void* realloc(void *ctx, void *ptr, size_t new_size)`	메모리 블록을 할당하거나 크기 조정합니다
`void free(void *ctx, void *ptr)`	메모리 블록을 해제합니다

버전 3.5에서 변경: `PyMemAllocator` 구조체의 이름이 *`PyMemAllocatorEx`*로 바뀌고 새로운 `calloc` 필드가 추가되었습니다.

type **PyMemAllocatorDomain**

할당자 도메인을 식별하는 데 사용되는 열거형. 도메인:

PYMEM_DOMAIN_RAW

함수:

- *`PyMem_RawMalloc()`*
- *`PyMem_RawRealloc()`*
- *`PyMem_RawCalloc()`*
- *`PyMem_RawFree()`*

PYMEM_DOMAIN_MEM

함수:

- *`PyMem_Malloc()`*,
- *`PyMem_Realloc()`*
- *`PyMem_Calloc()`*
- *`PyMem_Free()`*

PYMEM_DOMAIN_OBJ

함수:

- *`PyObject_Malloc()`*
- *`PyObject_Realloc()`*
- *`PyObject_Calloc()`*
- *`PyObject_Free()`*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

지정된 도메인의 메모리 블록 할당자를 가져옵니다.

void **PyMem_SetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

지정된 도메인의 메모리 블록 할당자를 설정합니다.

새 할당자는 0바이트를 요청할 때 고유한 `NULL`이 아닌 포인터를 반환해야 합니다.

For the *`PYMEM_DOMAIN_RAW`* domain, the allocator must be thread-safe: a *thread state* is not *attached* when the allocator is called.

For the remaining domains, the allocator must also be thread-safe: the allocator may be called in different interpreters that do not share a *GIL*.

새 할당자가 훅이 아니면 (이전 할당자를 호출하지 않으면), *`PyMem_SetupDebugHooks()`* 함수를 호출하여 새 할당자 위에 디버그 훅을 다시 설치해야 합니다.

See also *`PyPreConfig.allocator`* and *Preinitialize Python with PyPreConfig*.

경고

`PyMem_SetAllocator()` does have the following contract:

- It can be called after *`Py_PreInitialize()`* and before *`Py_InitializeFromConfig()`* to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without an *attached thread state*). See *the section on allocator domains* for more information.

- If called after Python has finish initializing (after *Py_InitializeFromConfig()* has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

버전 3.12에서 변경: All allocators must be thread-safe.

void **PyMem_SetupDebugHooks**(void)

Setup *debug hooks in the Python memory allocators* to detect memory errors.

12.8 파이썬 메모리 할당자의 디버그 훅

When Python is built in debug mode, the *PyMem_SetupDebugHooks()* function is called at the *Python preinitialization* to setup debug hooks on Python memory allocators to detect memory errors.

PYTHONMALLOC 환경 변수를 사용하여 릴리스 모드에서 컴파일된 파이썬에 디버그 훅을 설치할 수 있습니다 (예: PYTHONMALLOC=debug).

The *PyMem_SetupDebugHooks()* function can be used to set debug hooks after calling *PyMem_SetAllocator()*.

이 디버그 훅은 동적으로 할당된 메모리 블록을 특수하고, 인식 가능한 비트 패턴으로 채웁니다. 새로 할당된 메모리는 바이트 0xCD(PYMEM_CLEANBYTE)로 채워지고, 해제된 메모리는 바이트 0xDD(PYMEM_DEADBYTE)로 채워집니다. 메모리 블록은 바이트 0xFD (PYMEM_FORBIDDENBYTE)로 채워진 "금지된 바이트"로 둘러싸입니다. 이 바이트들의 문자열은 유효한 주소, 부동 소수점 또는 ASCII 문자열일 가능성이 낮습니다.

실행 시간 검사:

- API 위반 탐지. 예를 들어, *PyMem_Malloc()*이 할당한 메모리 블록에 *PyObject_Free()*가 호출되었는지 탐지합니다.
- 버퍼 시작 전에 쓰기 감지 (버퍼 언더플로).
- 버퍼 끝 뒤에 쓰기 감지 (버퍼 오버플로).
- Check that there is an *attached thread state* when allocator functions of *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) and *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) domains are called.

에러가 발생하면, 디버그 훅은 tracemalloc 모듈을 사용하여 메모리 블록이 할당된 곳의 트레이스백을 가져옵니다. tracemalloc이 파이썬 메모리 할당을 추적 중이고 메모리 블록이 추적될 때만 트레이스백이 표시됩니다.

Let *S* = `sizeof(size_t)`. `2*S` bytes are added at each end of each block of *N* bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (`p[i:j]` means the slice of bytes from `*(p+i)` inclusive up to `*(p+j)` exclusive; note that the treatment of negative indices differs from a Python slice):

p[-2*S:-S]
Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

p[-S]
API identifier (ASCII character):

- `'r'` for *PYMEM_DOMAIN_RAW*.
- `'m'` for *PYMEM_DOMAIN_MEM*.
- `'o'` for *PYMEM_DOMAIN_OBJ*.

p[-S+1:0]
Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

p[0:N]
The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with

PYMEM_DEADBYTE, to catch reference to freed memory. When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

p[N:N+S]
Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

p[N+S:N+2*S]
Only used if the `PYMEM_DEBUG_SERIALNO` macro is defined (not defined by default).

A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian `size_t`. If "bad memory" is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they' ve been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and tries to use it as an address. If you get in a debugger then and look at the object, you' re likely to see that it' s entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

버전 3.6에서 변경: The *PyMem_SetupDebugHooks()* function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if there is an *attached thread state* when functions of *PYMEM_DOMAIN_OBJ* and *PYMEM_DOMAIN_MEM* domains are called.

버전 3.8에서 변경: 바이트 패턴 `0xCB`(`PYMEM_CLEANBYTE`), `0xDB`(`PYMEM_DEADBYTE`) 및 `0xFB`(`PYMEM_FORBIDDENBYTE`)는 윈도우 CRT 디버그 `malloc()` 및 `free()`와 같은 값을 사용하도록 `0xCD`, `0xDD` 및 `0xFD`로 대체되었습니다.

12.9 pymalloc 할당자

파이썬에는 수명이 짧은 작은 (512바이트 이하) 객체에 최적화된 *pymalloc* 할당자가 있습니다. 32-비트 플랫폼에서는 256 KiB, 64-비트 플랫폼에서는 1 MiB의 고정 크기를 갖는 "아레나(arena)"라는 메모리 매핑을 사용합니다. 512 바이트보다 큰 할당의 경우 *PyMem_RawMalloc()*과 *PyMem_RawRealloc()*으로 대체됩니다.

*pymalloc*은 *PYMEM_DOMAIN_MEM*(예: *PyMem_Malloc()*)과 *PYMEM_DOMAIN_OBJ*(예: *PyObject_Malloc()*) 도메인의 기본 할당자입니다.

아레나 할당자는 다음 함수를 사용합니다:

- 윈도우에서 `VirtualAlloc()`과 `VirtualFree()`,
- 사용할 수 있으면 `mmap()`과 `munmap()`
- 그렇지 않으면 `malloc()`과 `free()`

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex: `PYTHONMALLOC=malloc`).

Typically, it makes sense to disable the pymalloc allocator when building Python with AddressSanitizer (`--with-address-sanitizer`) which helps uncover low level bugs within the C code.

12.9.1 pymalloc 아레나 할당자 사용자 정의

Added in version 3.4.

type **PyObjectArenaAllocator**
아레나 할당자를 기술하는 데 사용되는 구조체. 이 구조체에는 세 개의 필드가 있습니다:

필드	의미
`void *ctx`	첫 번째 인자로 전달된 사용자 컨텍스트
`void* alloc(void *ctx, size_t size)`	size 바이트의 아레나를 할당합니다
`void free(void *ctx, void *ptr, size_t size)`	아레나를 해제합니다

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

아레나 할당자를 얻습니다.

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

아레나 할당자를 설정합니다.

12.10 mimalloc 할당자

Added in version 3.13.

Python supports the mimalloc allocator when the underlying platform support is available. mimalloc "is a general purpose allocator with excellent performance characteristics. Initially developed by Daan Leijen for the runtime systems of the Koka and Lean languages."

12.11 tracemalloc C API

Added in version 3.7.

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

`tracemalloc` 모듈에서 할당된 메모리 블록을 추적합니다.

성공하면 `0`을 반환하고, 에러가 발생하면 (추적을 저장하기 위한 메모리를 할당하지 못했습니다) `-1`을 반환합니다. tracemalloc이 비활성화되었으면 `-2`를 반환합니다.

메모리 블록이 이미 추적되면, 기존 추적을 갱신합니다.

int **PyTraceMalloc_Untrack**(unsigned int domain, uintptr_t ptr)

`tracemalloc` 모듈에서 할당된 메모리 블록을 추적 해제합니다. 블록이 추적되지 않으면 아무것도 하지 않습니다.

tracemalloc이 비활성화되었으면 `-2`를 반환하고, 그렇지 않으면 `0`을 반환합니다.

12.12 예

다음은 개요 섹션에서 따온 예제입니다. I/O 버퍼가 첫 번째 함수 집합을 사용하여 파이썬 힙에서 할당되도록 다시 작성되었습니다:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* I/O 용 */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...buf 를 사용하는 I/O 연산을 수행합니다... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* PyMem_Malloc 으로 할당했습니다 */
return res;
```

형 지향 함수 집합을 사용하는 같은 코드입니다:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_New */
return res;
```

위의 두 가지 예에서, 버퍼는 항상 같은 집합에 속하는 함수를 통해 조작됨에 유의하십시오. 실제로, 서로 다른 할당자를 혼합할 위험이 최소로 줄어들도록, 주어진 메모리 블록에 대해 같은 메모리 API 패밀리를 사용하는 것은 필수입니다. 다음 코드 시퀀스에는 두 개의 에러가 있으며, 그중 하나는 서로 다른 힙에서 작동하는 두 개의 다른 할당자를 혼합하기 때문에 치명적(*fatal*)인 것으로 표시됩니다.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Free()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with *PyObject_New*, *PyObject_NewVar* and *PyObject_Free()*.

이것들은 C로 새로운 객체 형을 정의하고 구현하는 것에 대한 다음 장에서 설명될 것입니다.

CHAPTER 13

객체 구현 지원

이 장에서는 새 객체 형을 정의할 때 사용되는 함수, 형 및 매크로에 관해 설명합니다.

13.1 힙에 객체 할당하기

PyObject ***_PyObject_New**(*PyTypeObject* *type)

반환값: 새 참조.

PyVarObject ***_PyObject_NewVar**(*PyTypeObject* *type, *Py_ssize_t* size)

반환값: 새 참조.

PyObject ***PyObject_Init**(*PyObject* *op, *PyTypeObject* *type)

반환값: 빌린 참조. *Part of the* 안정 ABI. Initialize a newly allocated object *op* with its type and initial reference. Returns the initialized object. Other fields of the object are not initialized. Despite its name, this function is unrelated to the object's `__init__()` method (`tp_init` slot). Specifically, this function does **not** call the object's `__init__()` method.

In general, consider this function to be a low-level routine. Use `tp_alloc` where possible. For implementing `tp_alloc` for your type, prefer `PyType_GenericAlloc()` or `PyObject_New()`.

> **참고**
>
> This function only initializes the object's memory corresponding to the initial `PyObject` structure. It does not zero the rest.

PyVarObject ***PyObject_InitVar**(*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

반환값: 빌린 참조. *Part of the* 안정 ABI. 이것은 `PyObject_Init()`가 수행하는 모든 작업을 수행하고, 가변 크기 객체의 길이 정보도 초기화합니다.

> **참고**
>
> This function only initializes some of the object's memory. It does not zero the rest.

PyObject_New(TYPE, typeobj)

Allocates a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`) by calling *PyObject_Malloc()* to allocate memory and initializing it like *PyObject_Init()*. The caller will own the only reference to the object (i.e. its reference count will be one).

Avoid calling this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

This macro does not call *tp_alloc*, *tp_new* (`__new__()`), or *tp_init* (`__init__()`).

This cannot be used for objects with *Py_TPFLAGS_HAVE_GC* set in *tp_flags*; use *PyObject_GC_New* instead.

Memory allocated by this macro must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

참고

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

참고

This macro does not construct a fully initialized object of the given type; it merely allocates memory and prepares it for further initialization by *tp_init*. To construct a fully initialized object, call *typeobj* instead. For example:

```
PyObject *foo = PyObject_CallNoArgs((PyObject *)&PyFoo_Type);
```

더 보기

- *PyObject_Free()*
- *PyObject_GC_New*
- *PyType_GenericAlloc()*
- *tp_alloc*

PyObject_NewVar(TYPE, typeobj, size)

Like *PyObject_New* except:

- It allocates enough memory for the *TYPE* structure plus *size* (`Py_ssize_t`) fields of the size given by the *tp_itemsize* field of *typeobj*.
- The memory is initialized like *PyObject_InitVar()*.

This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

Avoid calling this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

This cannot be used for objects with *Py_TPFLAGS_HAVE_GC* set in *tp_flags*; use *PyObject_GC_NewVar* instead.

Memory allocated by this function must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

참고

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

참고

This macro does not construct a fully initialized object of the given type; it merely allocates memory and prepares it for further initialization by *tp_init*. To construct a fully initialized object, call *typeobj* instead. For example:

```
PyObject *list_instance = PyObject_CallNoArgs((PyObject *)&PyList_Type);
```

더 보기

- *PyObject_Free()*
- *PyObject_GC_NewVar*
- *PyType_GenericAlloc()*
- *tp_alloc*

void **PyObject_Del**(void *op)

Same as *PyObject_Free()*.

PyObject **_Py_NoneStruct**

파이썬에서 None으로 노출되는 객체. 이 객체에 대한 포인터로 평가되는 *Py_None* 매크로를 사용해서 액세스해야 합니다.

더 보기

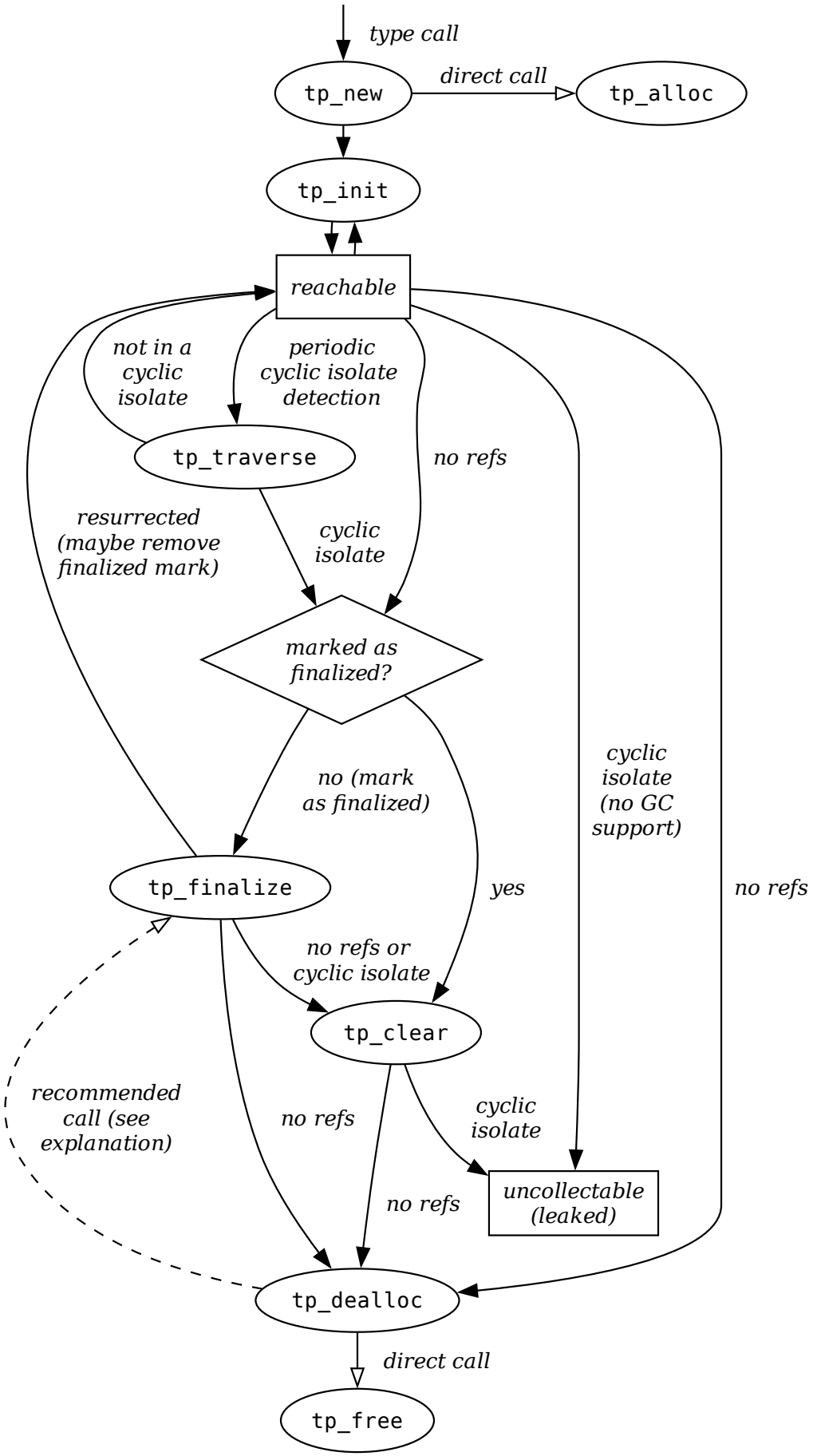
모듈 객체
확장 모듈을 할당하고 만듭니다.

13.2 Object Life Cycle

This section explains how a type's slots relate to each other throughout the life of an object. It is not intended to be a complete canonical reference for the slots; instead, refer to the slot-specific documentation in *Type Object Structures* for details about a particular slot.

13.2.1 Life Events

The figure below illustrates the order of events that can occur throughout an object's life. An arrow from *A* to *B* indicates that event *B* can occur after event *A* has occurred, with the arrow's label indicating the condition that must be true for *B* to occur after *A*.


type call
tp_new
direct call
tp_alloc
tp_init
reachable
not in a cyclic isolate
periodic cyclic isolate detection
tp_traverse
no refs
resurrected (maybe remove finalized mark)
cyclic isolate
marked as finalized?
cyclic isolate (no GC support)
no (mark as finalized)
tp_finalize
yes
no refs
no refs or cyclic isolate
tp_clear
recommended call (see explanation)
no refs
cyclic isolate
uncollectable (leaked)
no refs
tp_dealloc
direct call
tp_free

Explanation:

- When a new object is constructed by calling its type:
  1. *tp_new* is called to create a new object.
  2. *tp_alloc* is directly called by *tp_new* to allocate the memory for the new object.
  3. *tp_init* initializes the newly created object. `tp_init` can be called again to re-initialize an object, if desired. The `tp_init` call can also be skipped entirely, for example by Python code calling `__new__()`.
- After `tp_init` completes, the object is ready to use.
- Some time after the last reference to an object is removed:
  1. If an object is not marked as *finalized*, it might be finalized by marking it as *finalized* and calling its *tp_finalize* function. Python does *not* finalize an object when the last reference to it is deleted; use *PyObject_CallFinalizerFromDealloc()* to ensure that *tp_finalize* is always called.
  2. If the object is marked as finalized, *tp_clear* might be called by the garbage collector to clear references held by the object. It is *not* called when the object' s reference count reaches zero.
  3. *tp_dealloc* is called to destroy the object. To avoid code duplication, *tp_dealloc* typically calls into *tp_clear* to free up the object' s references.
  4. When *tp_dealloc* finishes object destruction, it directly calls *tp_free* (usually set to *PyObject_Free()* or *PyObject_GC_Del()* automatically as appropriate for the type) to deallocate the memory.
- The *tp_finalize* function is permitted to add a reference to the object if desired. If it does, the object is *resurrected*, preventing its pending destruction. (Only `tp_finalize` is allowed to resurrect an object; *tp_clear* and *tp_dealloc* cannot without calling into `tp_finalize`.) Resurrecting an object may or may not cause the object' s *finalized* mark to be removed. Currently, Python does not remove the *finalized* mark from a resurrected object if it supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set) but does remove the mark if the object does not support garbage collection; either or both of these behaviors may change in the future.
- *tp_dealloc* can optionally call *tp_finalize* via *PyObject_CallFinalizerFromDealloc()* if it wishes to reuse that code to help with object destruction. This is recommended because it guarantees that `tp_finalize` is always called before destruction. See the *tp_dealloc* documentation for example code.
- If the object is a member of a *cyclic isolate* and either *tp_clear* fails to break the reference cycle or the cyclic isolate is not detected (perhaps `gc.disable()` was called, or the *Py_TPFLAGS_HAVE_GC* flag was erroneously omitted in one of the involved types), the objects remain indefinitely uncollectable (they "leak"). See `gc.garbage`.

If the object is marked as supporting garbage collection (the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*), the following events are also possible:

- The garbage collector occasionally calls *tp_traverse* to identify *cyclic isolates*.
- When the garbage collector discovers a *cyclic isolate*, it finalizes one of the objects in the group by marking it as *finalized* and calling its *tp_finalize* function, if it has one. This repeats until the cyclic isolate doesn' t exist or all of the objects have been finalized.
- *tp_finalize* is permitted to resurrect the object by adding a reference from outside the *cyclic isolate*. The new reference causes the group of objects to no longer form a cyclic isolate (the reference cycle may still exist, but if it does the objects are no longer isolated).
- When the garbage collector discovers a *cyclic isolate* and all of the objects in the group have already been marked as *finalized*, the garbage collector clears one or more of the uncleared objects in the group (possibly concurrently) by calling each' s *tp_clear* function. This repeats as long as the cyclic isolate still exists and not all of the objects have been cleared.

13.2.2 Cyclic Isolate Destruction

Listed below are the stages of life of a hypothetical *cyclic isolate* that continues to exist after each member object is finalized or cleared. It is a memory leak if a cyclic isolate progresses through all of these stages; it should vanish once all objects are cleared, if not sooner. A cyclic isolate can vanish either because the reference cycle is broken or because the objects are no longer isolated due to finalizer resurrection (see `tp_finalize`).

0. **Reachable** (not yet a cyclic isolate): All objects are in their normal, reachable state. A reference cycle could exist, but an external reference means the objects are not yet isolated.
1. **Unreachable but consistent:** The final reference from outside the cyclic group of objects has been removed, causing the objects to become isolated (thus a cyclic isolate is born). None of the group' s objects have been finalized or cleared yet. The cyclic isolate remains at this stage until some future run of the garbage collector (not necessarily the next run because the next run might not scan every object).
2. **Mix of finalized and not finalized:** Objects in a cyclic isolate are finalized one at a time, which means that there is a period of time when the cyclic isolate is composed of a mix of finalized and non-finalized objects. Finalization order is unspecified, so it can appear random. A finalized object must behave in a sane manner when non-finalized objects interact with it, and a non-finalized object must be able to tolerate the finalization of an arbitrary subset of its referents.
3. **All finalized:** All objects in a cyclic isolate are finalized before any of them are cleared.
4. **Mix of finalized and cleared:** The objects can be cleared serially or concurrently (but with the *GIL* held); either way, some will finish before others. A finalized object must be able to tolerate the clearing of a subset of its referents. **PEP 442** calls this stage "cyclic trash".
5. **Leaked:** If a cyclic isolate still exists after all objects in the group have been finalized and cleared, then the objects remain indefinitely uncollectable (see `gc.garbage`). It is a bug if a cyclic isolate reaches this stage—it means the `tp_clear` methods of the participating objects have failed to break the reference cycle as required.

If `tp_clear` did not exist, then Python would have no way to safely break a reference cycle. Simply destroying an object in a cyclic isolate would result in a dangling pointer, triggering undefined behavior when an object referencing the destroyed object is itself destroyed. The clearing step makes object destruction a two-phase process: first `tp_clear` is called to partially destroy the objects enough to detangle them from each other, then `tp_dealloc` is called to complete the destruction.

Unlike clearing, finalization is not a phase of destruction. A finalized object must still behave properly by continuing to fulfill its design contracts. An object' s finalizer is allowed to execute arbitrary Python code, and is even allowed to prevent the impending destruction by adding a reference. The finalizer is only related to destruction by call order—if it runs, it runs before destruction, which starts with `tp_clear` (if called) and concludes with `tp_dealloc`.

The finalization step is not necessary to safely reclaim the objects in a cyclic isolate, but its existence makes it easier to design types that behave in a sane manner when objects are cleared. Clearing an object might necessarily leave it in a broken, partially destroyed state—it might be unsafe to call any of the cleared object' s methods or access any of its attributes. With finalization, only finalized objects can possibly interact with cleared objects; non-finalized objects are guaranteed to interact with only non-cleared (but potentially finalized) objects.

To summarize the possible interactions:

- A non-finalized object might have references to or from non-finalized and finalized objects, but not to or from cleared objects.
- A finalized object might have references to or from non-finalized, finalized, and cleared objects.
- A cleared object might have references to or from finalized and cleared objects, but not to or from non-finalized objects.

Without any reference cycles, an object can be simply destroyed once its last reference is deleted; the finalization and clearing steps are not necessary to safely reclaim unused objects. However, it can be useful to automatically call `tp_finalize` and `tp_clear` before destruction anyway because type design is simplified when all objects always experience the same series of events regardless of whether they participated in a cyclic isolate. Python currently only calls `tp_finalize` and `tp_clear` as needed to destroy a cyclic isolate; this may change in a future version.

13.2.3 Functions

To allocate and free memory, see 힙에 객체 할당하기.

void **PyObject_CallFinalizer**(*PyObject* *op)

Finalizes the object as described in *tp_finalize*. Call this function (or *PyObject_CallFinalizerFromDealloc()*) instead of calling *tp_finalize* directly because this function may deduplicate multiple calls to `tp_finalize`. Currently, calls are only deduplicated if the type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set); this may change in the future.

int **PyObject_CallFinalizerFromDealloc**(*PyObject* *op)

Same as *PyObject_CallFinalizer()* but meant to be called at the beginning of the object's destructor (*tp_dealloc*). There must not be any references to the object. If the object's finalizer resurrects the object, this function returns -1; no further destruction should happen. Otherwise, this function returns 0 and destruction can continue normally.

더 보기

tp_dealloc for example code.

13.3 공통 객체 구조체

파이썬의 객체 형 정의에 사용되는 많은 구조체가 있습니다. 이 섹션에서는 이러한 구조체와 사용 방법에 관해 설명합니다.

13.3.1 기본 객체 형과 매크로

모든 파이썬 객체는 궁극적으로 객체의 메모리 표현의 처음에서 적은 수의 필드를 공유합니다. 이들은 *PyObject*와 *PyVarObject* 형으로 표시되며, 다른 모든 파이썬 객체의 정의에서, 직접 또는 간접적으로, 사용되는 일부 매크로의 확장을 통해 정의됩니다. 추가 매크로는 참조 횟수에서 찾을 수 있습니다.

type **PyObject**

Part of the 제한된 API. (*Only some members are part of the stable ABI.*) All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a *PyObject*, but every pointer to a Python object can be cast to a *PyObject**.

The members must not be accessed directly; instead use macros such as *Py_REFCNT* and *Py_TYPE*.

Py_ssize_t **ob_refcnt**

Part of the 안정 ABI. The object's reference count, as returned by *Py_REFCNT*. Do not use this field directly; instead use functions and macros such as `Py_REFCNT`, *Py_INCREF()* and *Py_DecRef()*.

The field type may be different from `Py_ssize_t`, depending on build configuration and platform.

PyTypeObject ***ob_type**

Part of the 안정 ABI. The object's type. Do not use this field directly; use *Py_TYPE* and *Py_SET_TYPE()* instead.

type **PyVarObject**

Part of the 제한된 API. (*Only some members are part of the stable ABI.*) An extension of *PyObject* that adds the *ob_size* field. This is intended for objects that have some notion of *length*.

As with `PyObject`, the members must not be accessed directly; instead use macros such as *Py_SIZE*, *Py_REFCNT* and *Py_TYPE*.

Py_ssize_t **ob_size**

Part of the 안정 ABI. A size field, whose contents should be considered an object's internal implementation detail.

Do not use this field directly; use *Py_SIZE* instead.

Object creation functions such as *PyObject_NewVar()* will generally set this field to the requested size (number of items). After creation, arbitrary values can be stored in `ob_size` using *Py_SET_SIZE*.

To get an object's publicly exposed length, as returned by the Python function `len()`, use *PyObject_Length()* instead.

PyObject_HEAD

길이가 변하지 않는 객체를 나타내는 새로운 형을 선언할 때 사용되는 매크로입니다. PyObject_HEAD 매크로는 다음과 같이 확장됩니다:

```
PyObject ob_base;
```

위의 *PyObject* 설명서를 참조하십시오.

PyObject_VAR_HEAD

인스턴스마다 길이가 다른 객체를 나타내는 새로운 형을 선언할 때 사용되는 매크로입니다. PyObject_VAR_HEAD 매크로는 다음과 같이 확장됩니다:

```
PyVarObject ob_base;
```

위의 *PyVarObject* 설명서를 참조하십시오.

PyTypeObject **PyBaseObject_Type**

Part of the 안정 ABI. The base class of all other objects, the same as `object` in Python.

int **Py_Is**(*PyObject* *x, *PyObject* *y)

Part of the 안정 ABI 버전 *3.10* 이후로. Test if the *x* object is the *y* object, the same as `x is y` in Python.

Added in version 3.10.

int **Py_IsNone**(*PyObject* *x)

Part of the 안정 ABI 버전 *3.10* 이후로. Test if an object is the `None` singleton, the same as `x is None` in Python.

Added in version 3.10.

int **Py_IsTrue**(*PyObject* *x)

Part of the 안정 ABI 버전 *3.10* 이후로. Test if an object is the `True` singleton, the same as `x is True` in Python.

Added in version 3.10.

int **Py_IsFalse**(*PyObject* *x)

Part of the 안정 ABI 버전 *3.10* 이후로. Test if an object is the `False` singleton, the same as `x is False` in Python.

Added in version 3.10.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

반환값: 빌린 참조. *Part of the* 안정 ABI 버전 *3.14* 이후로. Get the type of the Python object *o*.

The returned reference is *borrowed* from *o*. Do not release it with *Py_DECREF()* or similar.

버전 3.11에서 변경: *Py_TYPE()* is changed to an inline static function. The parameter type is no longer `const PyObject*`.

int **Py_IS_TYPE**(*PyObject* *o, *PyTypeObject* *type)

객체 *o*의 형이 *type*이면 0이 아닌 값을 반환합니다. 그렇지 않으면 0을 반환합니다. `Py_TYPE(o) == type`과 동등합니다.

Added in version 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Set the type of object *o* to *type*, without any checking or reference counting.

This is a very low-level operation. Consider instead setting the Python attribute `__class__` using `PyObject_SetAttrString()` or similar.

Note that assigning an incompatible type can lead to undefined behavior.

If *type* is a *heap type*, the caller must create a new reference to it. Similarly, if the old type of *o* is a heap type, the caller must release a reference to that type.

Added in version 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the `ob_size` field of *o*.

버전 3.11에서 변경: `Py_SIZE()` is changed to an inline static function. The parameter type is no longer `const PyVarObject*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the `ob_size` field of *o* to *size*.

Added in version 3.9.

PyObject_HEAD_INIT(type)

이것은 새로운 `PyObject` 형의 초기화 값으로 확장되는 매크로입니다. 이 매크로는 다음으로 확장됩니다:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)

이것은 `ob_size` 필드를 포함하여, 새로운 `PyVarObject` 형의 초기화 값으로 확장되는 매크로입니다. 이 매크로는 다음으로 확장됩니다:

```
_PyObject_EXTRA_INIT
1, type, size,
```

13.3.2 함수와 메서드 구현

type **PyCFunction**

Part of the 안정 ABI. 대부분 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형. 이 형의 함수는 두 개의 `PyObject*` 매개 변수를 취하고 하나의 값을 반환합니다. 반환 값이 *NULL*이면, 예외가 설정되어 있어야 합니다. `NULL`이 아니면, 반환 값은 파이썬에 노출된 함수의 반환 값으로 해석됩니다. 함수는 새로운 참조를 반환해야 합니다.

함수 서명은 다음과 같습니다:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

Part of the 안정 ABI. 서명이 *METH_VARARGS | METH_KEYWORDS* 인 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형. 함수 서명은 다음과 같습니다:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type **PyCFunctionFast**

Part of the 안정 ABI 버전 *3.13* 이후로. 서명이 `METH_FASTCALL` 인 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형. 함수 서명은 다음과 같습니다:

```
PyObject *PyCFunctionFast(PyObject *self,
                          PyObject *const *args,
                          Py_ssize_t nargs);
```

type **PyCFunctionFastWithKeywords**

Part of the 안정 ABI 버전 *3.13* 이후로. 서명이 *METH_FASTCALL | METH_KEYWORDS* 인 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형. 함수 서명은 다음과 같습니다:

```
PyObject *PyCFunctionFastWithKeywords(PyObject *self,
                                      PyObject *const *args,
                                      Py_ssize_t nargs,
                                      PyObject *kwnames);
```

type **PyCMethod**

서명이 *METH_METHOD | METH_FASTCALL | METH_KEYWORDS* 인 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형. 함수 서명은 다음과 같습니다:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Added in version 3.9.

type **PyMethodDef**

Part of the 안정 ABI (*including all members*). 확장형의 메서드를 기술하는 데 사용되는 구조체. 이 구조체에는 네 개의 필드가 있습니다:

const char ***ml_name**

메서드의 이름.

PyCFunction **ml_meth**

C 구현에 대한 포인터.

int **ml_flags**

호출 구성 방법을 나타내는 플래그 비트.

const char ***ml_doc**

독스트링의 내용을 가리킵니다.

*ml_meth*는 C 함수 포인터입니다. 함수는 형이 다를 수 있지만, 항상 *PyObject**를 반환합니다. 함수가 *PyCFunction*이 아니면, 컴파일러는 메서드 테이블에서 캐스트를 요구합니다. *PyCFunction*이 첫 번째 매개 변수를 *PyObject**로 정의하더라도, 일반적으로 메서드 구현은 *self* 객체의 특정 C 형을 사용합니다.

ml_flags 필드는 다음 플래그를 포함 할 수 있는 비트 필드입니다. 개별 플래그는 호출 규칙이나 바인딩 규칙을 나타냅니다.

다음과 같은 호출 규칙이 있습니다:

METH_VARARGS

이는 메서드가 *PyCFunction* 형인 일반적인 호출 규칙입니다. 함수는 두 개의 *PyObject** 값을 기대합니다. 첫 번째는 메서드의 *self* 객체입니다; 모듈 함수의 경우, 모듈 객체입니다. 두 번째 매개 변수(종종 *args*라고 합니다)는 모든 인자를 나타내는 튜플 객체입니다. 이 매개 변수는 일반적으로 *PyArg_ParseTuple()*이나 *PyArg_UnpackTuple()*을 사용하여 처리됩니다.

METH_KEYWORDS

Can only be used in certain combinations with other flags: *METH_VARARGS | METH_KEYWORDS*, *METH_FASTCALL | METH_KEYWORDS* and *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_VARARGS* | *METH_KEYWORDS
이러한 플래그가 있는 메서드는 *PyCFunctionWithKeywords* 형이어야 합니다. 이 함수는 세 개의 매개 변수를 기대합니다: *self*, *args*, *kwargs*. 여기서 *kwargs*는 모든 키워드 인자의 딕셔너리이거나 키워드 인자가 없으면 NULL일 수 있습니다. 매개 변수는 일반적으로 *PyArg_ParseTupleAndKeywords()*를 사용하여 처리됩니다.

METH_FASTCALL
위치 인자만 지원하는 빠른 호출 규칙. 메서드의 형은 *PyCFunctionFast* 입니다. 첫 번째 매개 변수는 *self*이고, 두 번째 매개 변수는 인자를 나타내는 *PyObject** 값의 C 배열이며, 세 번째 매개 변수는 인자 수(배열의 길이)입니다.

Added in version 3.7.

버전 3.10에서 변경: METH_FASTCALL은 이제 안정 *API*의 일부입니다.

METH_FASTCALL* | *METH_KEYWORDS
PyCFunctionFastWithKeywords 형의 메서드를 사용하여, 키워드 인자도 지원하는 *METH_FASTCALL*의 확장. 키워드 인자는 벡터콜(*vectorcall*) 프로토콜과 같은 방식으로 전달됩니다: 추가의 네 번째 *PyObject** 매개 변수가 있는데, 키워드 인자의 이름(문자열임이 보장됩니다)을 나타내는 튜플이거나 키워드가 없으면 NULL일 수 있습니다. 키워드 인자의 값은 위치 인자 다음에 *args* 배열에 저장됩니다.

Added in version 3.7.

METH_METHOD
Can only be used in the combination with other flags: *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS
정의하는 클래스(*defining class*), 즉, 문제의 메서드를 포함하는 클래스를 지원하는 *METH_FASTCALL | METH_KEYWORDS*의 확장. 정의하는 클래스는 Py_TYPE(self)의 슈퍼 클래스일 수 있습니다.

메서드는 *PyCMethod* 형이어야 하는데, self 뒤에 defining_class 인자가 추가된 METH_FASTCALL | METH_KEYWORDS와 같습니다.

Added in version 3.9.

METH_NOARGS
매개 변수가 없는 메서드는 *METH_NOARGS* 플래그로 나열되어 있으면, 인자가 주어졌는지 확인할 필요가 없습니다. *PyCFunction* 형이어야 합니다. 첫 번째 매개 변수의 이름은 일반적으로 *self*이며 모듈이나 객체 인스턴스에 대한 참조를 보유합니다. 모든 경우에 두 번째 매개 변수는 NULL입니다.

The function must have 2 parameters. Since the second parameter is unused, *Py_UNUSED* can be used to prevent a compiler warning.

METH_O
"O" 인자로 *PyArg_ParseTuple()*을 호출하는 대신, 단일 객체 인자가 있는 메서드는 *METH_O* 플래그로 나열 할 수 있습니다. *PyCFunction* 형이고, *self* 매개 변수와 단일 인자를 나타내는 *PyObject** 매개 변수를 갖습니다.

이 두 상수는 호출 규칙을 나타내는 데 사용되지 않고 클래스의 메서드와 함께 사용할 때 바인딩을 나타냅니다. 모듈에 정의된 함수에는 사용할 수 없습니다. 이러한 플래그 중 최대 하나를 주어진 메서드에 대해 설정할 수 있습니다.

METH_CLASS
메서드로 형의 인스턴스가 아닌 형 객체가 첫 번째 매개 변수로 전달됩니다. classmethod() 내장 함수를 사용할 때 만들어지는 것과 유사한 클래스 메서드(*class methods*)를 만드는 데 사용됩니다.

METH_STATIC
메서드로 형의 인스턴스가 아닌 NULL이 첫 번째 매개 변수로 전달됩니다. staticmethod() 내장 함수를 사용할 때 만들어지는 것과 유사한 정적 메서드(*static methods*)를 만드는 데 사용됩니다.

하나의 다른 상수는 같은 메서드 이름을 가진 다른 정의 대신 메서드가 로드되는지를 제어합니다.

METH_COEXIST

기존 정의 대신 메서드가 로드됩니다. *METH_COEXIST*가 없으면, 기본값은 반복되는 정의를 건너뛰는 것입니다. 슬롯 래퍼가 메서드 테이블 전에 로드되므로, 예를 들어 *sq_contains* 슬롯의 존재는 `__contains__()` 라는 래핑 된 메서드를 생성하고 같은 이름의 해당 PyCFunction을 로드하지 않습니다. 플래그가 정의되면, PyCFunction이 래퍼 객체 자리에 로드되고 슬롯과 공존합니다. 이는 PyCFunction에 대한 호출이 래퍼 객체 호출보다 최적화되어 있기 때문에 유용합니다.

PyObject ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.9* 이후로. Turn *ml* into a Python *callable* object. The caller must ensure that *ml* outlives the *callable*. Typically, *ml* is defined as a static variable.

The *self* parameter will be passed as the *self* argument to the C function in `ml->ml_meth` when invoked. *self* can be `NULL`.

The *callable* object' s `__module__` attribute can be set from the given *module* argument. *module* should be a Python string, which will be used as name of the module the function is defined in. If unavailable, it can be set to `None` or `NULL`.

더 보기

`function.__module__`

The *cls* parameter will be passed as the *defining_class* argument to the C function. Must be set if *METH_METHOD* is set on `ml->ml_flags`.

Added in version 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

반환값: 새 참조. *Part of the* 안정 ABI. Equivalent to `PyCMethod_New(ml, self, module, NULL)`.

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

반환값: 새 참조. *Part of the* 안정 ABI 버전 *3.4* 이후로. Equivalent to `PyCMethod_New(ml, self, NULL, NULL)`.

13.3.3 확장형의 어트리뷰트 액세스

type **PyMemberDef**

Part of the 안정 ABI (*including all members*). C 구조체 멤버에 해당하는 형의 어트리뷰트를 기술하는 구조체. 클래스를 정의할 때, 이 구조체들의 NULL-종료 배열을 *tp_members* 슬롯에 넣습니다.

필드는 순서대로 다음과 같습니다:

const char ***name**

Name of the member. A NULL value marks the end of a `PyMemberDef[]` array.

The string should be static, no copy is made of it.

int **type**

The type of the member in the C struct. See *Member types* for the possible values.

Py_ssize_t **offset**

멤버가 형의 객체 구조체에 위치하는 바이트 단위의 오프셋.

int **flags**

Zero or more of the *Member flags*, combined using bitwise OR.

const char ***doc**

The docstring, or NULL. The string should be static, no copy is made of it. Typically, it is defined using *PyDoc_STR*.

By default (when *flags* is `0`), members allow both read and write access. Use the *Py_READONLY* flag for read-only access. Certain types, like *Py_T_STRING*, imply *Py_READONLY*. Only *Py_T_OBJECT_EX* (and legacy *T_OBJECT*) members can be deleted.

For heap-allocated types (created using *PyType_FromSpec()* or similar), `PyMemberDef` may contain a definition for the special member `"__vectorcalloffset__"`, corresponding to *tp_vectorcall_offset* in type objects. This member must be defined with `Py_T_PYSSIZET`, and either `Py_READONLY` or `Py_READONLY | Py_RELATIVE_OFFSET`. For example:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

(You may need to `#include <stddef.h>` for `offsetof()`.)

The legacy offsets *tp_dictoffset* and *tp_weaklistoffset* can be defined similarly using `"__dictoffset__"` and `"__weaklistoffset__"` members, but extensions are strongly encouraged to use *Py_TPFLAGS_MANAGED_DICT* and *Py_TPFLAGS_MANAGED_WEAKREF* instead.

버전 3.12에서 변경: `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

버전 3.14에서 변경: *Py_RELATIVE_OFFSET* is now allowed for `"__vectorcalloffset__"`, `"__dictoffset__"` and `"__weaklistoffset__"`.

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

Part of the 안정 ABI. Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

버전 3.12에서 변경: `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

Part of the 안정 ABI. Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by `PyMemberDef` *m*. Returns `0` if successful and a negative value on failure.

버전 3.12에서 변경: `PyMember_SetOne` is always available. Previously, it required including `"structmember.h"`.

Member flags

The following flags can be used with *PyMemberDef.flags*:

Py_READONLY

Not writable.

Py_AUDIT_READ

Emit an `object.__getattr__` audit event before reading.

Py_RELATIVE_OFFSET

Indicates that the *offset* of this `PyMemberDef` entry indicates an offset from the subclass-specific data, rather than from `PyObject`.

Can only be used as part of *Py_tp_members slot* when creating a class using negative *basicsize*. It is mandatory in that case.

This flag is only used in *PyType_Slot*. When setting *tp_members* during class creation, Python clears it and sets *PyMemberDef.offset* to the offset from the `PyObject` struct.

버전 3.10에서 변경: The `RESTRICTED`, `READ_RESTRICTED` and `WRITE_RESTRICTED` macros available with `#include "structmember.h"` are deprecated. `READ_RESTRICTED` and `RESTRICTED` are equivalent to *Py_AUDIT_READ*; `WRITE_RESTRICTED` does nothing.

버전 3.12에서 변경: The `READONLY` macro was renamed to *`Py_READONLY`*. The `PY_AUDIT_READ` macro was renamed with the `Py_` prefix. The new names are now always available. Previously, these required `#include "structmember.h"`. The header is still available and it provides the old names.

Member types

`PyMemberDef.type` can be one of the following macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type. When it is set from Python, it will be converted back to the C type. If that is not possible, an exception such as `TypeError` or `ValueError` is raised.

Unless marked (D), attributes defined this way cannot be deleted using e.g. `del` or `delattr()`.

매크로 이름	C 형	Python type
Py_T_BYTE	`char`	`int`
Py_T_SHORT	`short`	`int`
Py_T_INT	`int`	`int`
Py_T_LONG	`long`	`int`
Py_T_LONGLONG	`long long`	`int`
Py_T_UBYTE	`unsigned char`	`int`
Py_T_UINT	`unsigned int`	`int`
Py_T_USHORT	`unsigned short`	`int`
Py_T_ULONG	`unsigned long`	`int`
Py_T_ULONGLONG	`unsigned long long`	`int`
Py_T_PYSSIZET	*Py_ssize_t*	`int`
Py_T_FLOAT	`float`	`float`
Py_T_DOUBLE	`double`	`float`
Py_T_BOOL	`char` (written as 0 or 1)	`bool`
Py_T_STRING	`const char*` (*)	`str` (RO)
Py_T_STRING_INPLACE	`const char[]` (*)	`str` (RO)
Py_T_CHAR	`char` (0-127)	`str` (**)
Py_T_OBJECT_EX	*PyObject**	`object` (D)

(*): Zero-terminated, UTF8-encoded C string. With `Py_T_STRING` the C representation is a pointer; with `Py_T_STRING_INPLACE` the string is stored directly in the structure.

(**): String of length 1. Only ASCII is accepted.

(RO): Implies *Py_READONLY*.

(D): Can be deleted, in which case the pointer is set to NULL. Reading a NULL pointer raises AttributeError.

Added in version 3.12: In previous versions, the macros were only available with #include "structmember.h" and were named without the Py_ prefix (e.g. as T_INT). The header is still available and contains the old names, along with the following deprecated types:

T_OBJECT

Like Py_T_OBJECT_EX, but NULL is converted to None. This results in surprising behavior in Python: deleting the attribute effectively sets it to None.

T_NONE

Always None. Must be used with *Py_READONLY*.

Defining Getters and Setters

type **PyGetSetDef**

Part of the 안정 ABI (*including all members*). 형에 대한 프로퍼티 같은 액세스를 정의하는 구조체. *PyTypeObject.tp_getset* 슬롯에 대한 설명도 참조하십시오.

const char ***name**

어트리뷰트 이름

getter **get**

어트리뷰트를 얻는 C 함수.

setter **set**

어트리뷰트를 설정하거나 삭제하는 선택적 C 함수. NULL이면, 어트리뷰트는 읽기 전용입니다.

const char ***doc**

선택적 독스트링

void ***closure**

선택적 사용자 데이터 포인터, getter와 setter에 추가 데이터를 제공합니다.

typedef *PyObject* *(***getter**)(*PyObject**, void*)

Part of the 안정 ABI. get 함수는 하나의 *PyObject** 매개 변수(인스턴스)와 사용자 데이터 포인터(연관된 closure)를 받아들입니다:

성공하면 새 참조를 반환하고, 실패하면 설정된 예외와 함께 NULL을 반환해야 합니다.

typedef int (***setter**)(*PyObject**, *PyObject**, void*)

Part of the 안정 ABI. set 함수는 두 개의 *PyObject** 매개 변수(인스턴스와 설정할 값)와 사용자 데이터 포인터(연관된 closure)를 받아들입니다:

어트리뷰트를 삭제해야 하는 경우 두 번째 매개 변수는 NULL입니다. 성공하면 0을, 실패하면 설정된 예외와 함께 -1을 반환해야 합니다.

13.4 Type Object Structures

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the *PyTypeObject* structure. Type objects can be handled using any of the PyObject_* or PyType_* functions, but do not offer much that' s interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

형 객체는 대부분 표준형보다 상당히 큽니다. 크기가 큰 이유는 각 형 객체가 많은 수의 값을 저장하기 때문인데, 주로 C 함수 포인터이고 각기 형의 기능 중 작은 부분을 구현합니다. 이 섹션에서는 형 객체의 필드를 자세히 살펴봅니다. 필드는 구조체에서 나타나는 순서대로 설명됩니다.

다음의 간략 참조 외에도, 예 섹션은 *PyTypeObject*의 의미와 사용에 대한 통찰을 제공합니다.

13.4.1 간략 참조

"tp 슬롯"

PyTypeObject 슬롯[Page 316, 1]	형	특수 메서드/어트리뷰트	정보[Page 316, 2]			
			C	T	D	I
<R> tp_name	const char *	__name__	X	X		
tp_basicsize	Py_ssize_t		X	X		X
tp_itemsize	Py_ssize_t			X		X
tp_dealloc	destructor		X	X		X
tp_vectorcall_offset	Py_ssize_t			X		X
(tp_getattr)	getattrfunc	__getattribute__, __getattr__				G
(tp_setattr)	setattrfunc	__setattr__, __delattr__				G
tp_as_async	PyAsyncMethods *	서브 슬롯				%
tp_repr	reprfunc	__repr__	X	X		X
tp_as_number	PyNumberMethods *	서브 슬롯				%
tp_as_sequence	PySequenceMethods *	서브 슬롯				%
tp_as_mapping	PyMappingMethods *	서브 슬롯				%
tp_hash	hashfunc	__hash__	X			G
tp_call	ternaryfunc	__call__		X		X
tp_str	reprfunc	__str__	X			X
tp_getattro	getattrofunc	__getattribute__, __getattr__	X	X		G
tp_setattro	setattrofunc	__setattr__, __delattr__	X	X		G
tp_as_buffer	PyBufferProcs *	서브 슬롯				%
tp_flags	unsigned long		X	X		?
tp_doc	const char *	__doc__	X	X		
tp_traverse	traverseproc			X		G
tp_clear	inquiry			X		G
tp_richcompare	richcmpfunc	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(tp_weaklistoffset)	Py_ssize_t			X		?
tp_iter	getiterfunc	__iter__				X
tp_iternext	iternextfunc	__next__				X
tp_methods	PyMethodDef []		X	X		
tp_members	PyMemberDef []			X		
tp_getset	PyGetSetDef []		X	X		
tp_base	PyTypeObject *	__base__			X	
tp_dict	PyObject *	__dict__			?	
tp_descr_get	descrgetfunc	__get__				X
tp_descr_set	descrsetfunc	__set__, __delete__				X
(tp_dictoffset)	Py_ssize_t			X		?
tp_init	initproc	__init__	X	X		X
tp_alloc	allocfunc		X		?	?
tp_new	newfunc	__new__	X	X	?	?
tp_free	freefunc		X	X	?	?
tp_is_gc	inquiry			X		X
<tp_bases>	PyObject *	__bases__			~	
<tp_mro>	PyObject *	__mro__			~	
[tp_cache]	PyObject *					
[tp_subclasses]	void *	__subclasses__				
[tp_weaklist]	PyObject *					
(tp_del)	destructor					
[tp_version_tag]	unsigned int					
tp_finalize	destructor	__del__				X
tp_vectorcall	vectorcallfunc					

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

PyTypeObject 슬롯[1]	형	특수 메서드/어트리뷰트	정보[2] C T D I
[*tp_watched*]	unsigned char		

서브 슬롯

슬롯	형	특수 메서드
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
nb_inplace_rshift	*binaryfunc*	__irshift__
nb_and	*binaryfunc*	__and__ __rand__

다음 페이지에 계속

[1] (): A slot name in parentheses indicates it is (effectively) deprecated.
<>: Names in angle brackets should be initially set to NULL and treated as read-only.
[]: Names in square brackets are for internal use only.
<R> (as a prefix) means the field is required (must be non-NULL).

[2] 열:
"O": set on *PyBaseObject_Type*
"T": set on *PyType_Type*
"D": 기본값 (슬롯이 NULL로 설정된 경우)

```
X - PyType_Ready sets this value if it is NULL
~ - PyType_Ready always sets this value (it should be NULL)
? - PyType_Ready may set this value depending on other slots

Also see the inheritance column ("I").
```

"I": 상속

```
X - type slot is inherited via *PyType_Ready* if defined with a *NULL* value
% - the slots of the sub-struct are inherited individually
G - inherited, but only in combination with other slots; see the slot's description
? - it's complicated; see the slot's description
```

일부 슬롯은 일반 어트리뷰트 조회 체인을 통해 효과적으로 상속됨에 유의하십시오.

표 2 – 이전 페이지에서 계속

슬롯	형	특수 메서드
nb_inplace_and	*binaryfunc*	__iand__
nb_xor	*binaryfunc*	__xor__ __rxor__
nb_inplace_xor	*binaryfunc*	__ixor__
nb_or	*binaryfunc*	__or__ __ror__
nb_inplace_or	*binaryfunc*	__ior__
nb_int	*unaryfunc*	__int__
nb_reserved	void *	
nb_float	*unaryfunc*	__float__
nb_floor_divide	*binaryfunc*	__floordiv__
nb_inplace_floor_divide	*binaryfunc*	__ifloordiv__
nb_true_divide	*binaryfunc*	__truediv__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	__buffer__
bf_releasebuffer	*releasebufferproc()*	__release_buffer__

슬롯 typedef

typedef	매개 변수 형	반환형
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc	*PyObject* * *visitproc* void *	int
newfunc	*PyTypeObject* * *PyObject* * *PyObject* *	*PyObject* *
initproc	*PyObject* * *PyObject* * *PyObject* *	int
reprfunc	*PyObject* *	*PyObject* *
getattrfunc	*PyObject* * const char *	*PyObject* *
setattrfunc	*PyObject* * const char * *PyObject* *	int
getattrofunc	*PyObject* * *PyObject* *	*PyObject* *
setattrofunc	*PyObject* * *PyObject* * *PyObject* *	int
descrgetfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
descrsetfunc	*PyObject* * *PyObject* * *PyObject* *	int
hashfunc	*PyObject* *	Py_hash_t
richcmpfunc		*PyObject* *

자세한 내용은 아래 슬롯 형 *typedef*를 참조하십시오.

13.4.2 PyTypeObject 정의

The structure definition for *PyTypeObject* can be found in `Include/cpython/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    PyMethodDef *tp_methods;
    PyMemberDef *tp_members;
    PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    PyTypeObject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache; /* no longer used */
    void *tp_subclasses;  /* for static builtin types this is an index */
    PyObject *tp_weaklist; /* not used for static builtin types */
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6.
     * If zero, the cache is invalid and must be initialized.
     */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;

    /* Number of tp_version_tag values used.
     * Set to _Py_ATTR_CACHE_UNUSED if the attribute cache is
     * disabled for this type (e.g. due to custom MRO entries).
     * Otherwise, limited to MAX_VERSIONS_PER_CLASS (defined elsewhere).
     */
    uint16_t tp_versions_used;
} PyTypeObject;
```

13.4.3 PyObject 슬롯

The type object structure extends the *PyVarObject* structure. The *ob_size* field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the *ob_size* field.

PyObject.ob_refcnt

> The type object' s reference count is initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for *statically allocated type objects*, the type' s instances (objects whose *ob_type* points back to the type) do *not* count as references. But for *dynamically allocated type objects*, the instances *do* count as references.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

PyObject.ob_type

이것은 형의 형, 즉 메타 형(metatype) 입니다. PyObject_HEAD_INIT 매크로에 대한 인자로 초기화되며, 값은 일반적으로 &PyType_Type 이어야 합니다. 그러나, (적어도) 윈도우에서 사용 가능해야 하는 동적으로 로드 가능한 확장 모듈의 경우, 컴파일러는 유효한 초기화자가 아니라고 불평합니다. 따라서, 규칙은 NULL을 PyObject_HEAD_INIT 매크로로 전달하고, 다른 작업을 수행하기 전에 모듈의 초기화 함수 시작에서 필드를 명시적으로 초기화하는 것입니다. 이것은 일반적으로 다음과 같이 수행됩니다:

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. *PyType_Ready()* checks if *ob_type* is NULL, and if so, initializes it to the *ob_type* field of the base class. *PyType_Ready()* will not change this field if it is non-zero.

계승:

이 필드는 서브 형으로 상속됩니다.

13.4.4 PyVarObject 슬롯

PyVarObject.ob_size

For *statically allocated type objects*, this should be initialized to zero. For *dynamically allocated type objects*, this field has a special internal meaning.

This field should be accessed using the *Py_SIZE()* macro.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

13.4.5 PyTypeObject 슬롯

Each slot has a section describing inheritance. If *PyType_Ready()* may set a value when the field is set to NULL then there will also be a "Default" section. (Note that many fields set on *PyBaseObject_Type* and *PyType_Type* effectively act as defaults.)

const char **PyTypeObject*.**tp_name**

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named T defined in module M in subpackage Q in package P should have the *tp_name* initializer "P.Q.M.T".

For *dynamically allocated type objects*, this should just be the type name, and the module name explicitly stored in the type dict as the value for key '__module__'.

For *statically allocated type objects*, the *tp_name* field should contain a dot. Everything before the last dot is made accessible as the __module__ attribute, and everything after the last dot is made accessible as the __name__ attribute.

If no dot is present, the entire *tp_name* field is made accessible as the __name__ attribute, and the __module__ attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

이 필드는 NULL이 아니어야 합니다. *PyTypeObject()*에서 유일하게 필요한 필드입니다 (잠재적인 *tp_itemsize*를 제외하고).

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

Py_ssize_t *PyTypeObject*.**tp_basicsize**

Py_ssize_t *PyTypeObject*.**tp_itemsize**

이 필드를 사용하면 형 인스턴스의 크기를 바이트 단위로 계산할 수 있습니다.

There are two kinds of types: types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`. (Exceptions to this rule can be made using *PyUnstable_Object_GC_NewWithExtraData()*.)

For a type with variable-length instances, the instances must have an *ob_size* field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the "length" of the object.

Functions like *PyObject_NewVar()* will take the value of N as an argument, and store in the instance' s *ob_size* field. Note that the *ob_size* field may later be used for other purposes. For example, `int` instances use the bits of *ob_size* in an implementation-defined way; the underlying storage and its size should be accessed using *PyLong_Export()*.

> **참고**
>
> The *ob_size* field should be accessed using the *Py_SIZE()* and *Py_SET_SIZE()* macros.

Also, the presence of an *ob_size* field in the instance layout doesn' t mean that the instance structure is variable-length. For example, the `list` type has fixed-length instances, yet those instances have a *ob_size* field. (As with `int`, avoid reading lists' `ob_size` directly. Call *PyList_Size()* instead.)

The `tp_basicsize` includes size needed for data of the type' s *tp_base*, plus any extra data needed by each instance.

The correct way to set `tp_basicsize` is to use the `sizeof` operator on the struct used to declare the instance layout. This struct must include the struct used to declare the base type. In other words, `tp_basicsize` must be greater than or equal to the base' s `tp_basicsize`.

Since every type is a subtype of `object`, this struct must include *PyObject* or *PyVarObject* (depending on whether *ob_size* should be included). These are usually defined by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD*, respectively.

The basic size does not include the GC header size, as that header is not part of *PyObject_HEAD*.

For cases where struct used to declare the base type is unknown, see *PyType_Spec.basicsize* and *PyType_FromMetaclass()*.

Notes about alignment:

- `tp_basicsize` must be a multiple of `_Alignof(PyObject)`. When using `sizeof` on a `struct` that includes *PyObject_HEAD*, as recommended, the compiler ensures this. When not using a C `struct`, or when using compiler extensions like `__attribute__((packed))`, it is up to you.
- If the variable items require a particular alignment, `tp_basicsize` and `tp_itemsize` must each be a multiple of that alignment. For example, if a type' s variable part stores a `double`, it is your responsibility that both fields are a multiple of `_Alignof(double)`.

계승:

These fields are inherited separately by subtypes. (That is, if the field is set to zero, *PyType_Ready()* will copy the value from the base type, indicating that the instances do not need additional storage.)

If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor *PyTypeObject*.**tp_dealloc**

A pointer to the instance destructor function. The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function should remove all references which the instance owns (e.g., call *Py_CLEAR()*), free all memory buffers owned by the instance, and call the type' s *tp_free* function to free the object itself.

If you may call functions that may set the error indicator, you must use *PyErr_GetRaisedException()* and *PyErr_SetRaisedException()* to ensure you don' t clobber a preexisting error indicator (the deallocation could have occurred while processing a different error):

```
static void
foo_dealloc(foo_object *self)
{
    PyObject *et, *ev, *etb;
    PyObject *exc = PyErr_GetRaisedException();
    ...
    PyErr_SetRaisedException(exc);
}
```

The dealloc handler itself must not raise an exception; if it hits an error case it should call *PyErr_FormatUnraisable()* to log (and clear) an unraisable exception.

No guarantees are made about when an object is destroyed, except:

- Python will destroy an object immediately or some time after the final reference to the object is deleted, unless its finalizer (*tp_finalize*) subsequently resurrects the object.
- An object will not be destroyed while it is being automatically finalized (*tp_finalize*) or automatically cleared (*tp_clear*).

CPython currently destroys an object immediately from *Py_DECREF()* when the new reference count is zero, but this may change in a future version.

It is recommended to call *PyObject_CallFinalizerFromDealloc()* at the beginning of `tp_dealloc` to guarantee that the object is always finalized before destruction.

If the type supports garbage collection (the *Py_TPFLAGS_HAVE_GC* flag is set), the destructor should call *PyObject_GC_UnTrack()* before clearing any member fields.

It is permissible to call *tp_clear* from `tp_dealloc` to reduce code duplication and to guarantee that the object is always cleared before destruction. Beware that `tp_clear` might have already been called.

If the type is heap allocated (*Py_TPFLAGS_HEAPTYPE*), the deallocator should release the owned reference to its type object (via *Py_DECREF()*) after calling the type deallocator. See the example code below.:

```
static void
foo_dealloc(PyObject *op)
{
   foo_object *self = (foo_object *) op;
   PyObject_GC_UnTrack(self);
   Py_CLEAR(self->ref);
   Py_TYPE(self)->tp_free(self);
}
```

`tp_dealloc` must leave the exception status unchanged. If it needs to call something that might raise an exception, the exception state must be backed up first and restored later (after logging any exceptions with *PyErr_WriteUnraisable()*).

Example:

```
static void
foo_dealloc(PyObject *self)
{
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    PyObject *exc = PyErr_GetRaisedException();

    if (PyObject_CallFinalizerFromDealloc(self) < 0) {
        // self was resurrected.
        goto done;
    }

    PyTypeObject *tp = Py_TYPE(self);

    if (tp->tp_flags & Py_TPFLAGS_HAVE_GC) {
        PyObject_GC_UnTrack(self);
    }

    // Optional, but convenient to avoid code duplication.
    if (tp->tp_clear && tp->tp_clear(self) < 0) {
        PyErr_WriteUnraisable(self);
    }

    // Any additional destruction goes here.

    tp->tp_free(self);
    self = NULL;  // In case PyErr_WriteUnraisable() is called below.

    if (tp->tp_flags & Py_TPFLAGS_HEAPTYPE) {
        Py_CLEAR(tp);
    }

done:
    // Optional, if something was called that might have raised an
    // exception.
    if (PyErr_Occurred()) {
        PyErr_WriteUnraisable(self);
    }
    PyErr_SetRaisedException(exc);
}
```

`tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called with an *attached thread state*. However, if the object being destroyed in turn destroys objects from some other C library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.

계승:

이 필드는 서브 형으로 상속됩니다.

더 보기

Object Life Cycle for details about how this slot relates to other slots.

Py_ssize_t *PyTypeObject*.**tp_vectorcall_offset**

간단한 *tp_call*의 더 효율적인 대안인 벡터콜(*vectorcall*) 프로토콜을 사용하여 객체를 호출하는 것을 구현하는 인스턴스별 함수에 대한 선택적 오프셋입니다.

This field is only used if the flag *Py_TPFLAGS_HAVE_VECTORCALL* is set. If so, this must be a positive integer containing the offset in the instance of a *vectorcallfunc* pointer.

The *vectorcallfunc* pointer may be NULL, in which case the instance behaves as if *Py_TPFLAGS_HAVE_VECTORCALL* was not set: calling the instance falls back to *tp_call*.

Py_TPFLAGS_HAVE_VECTORCALL을 설정하는 모든 클래스는 *tp_call*도 설정해야 하고, 해당 동작이 *vectorcallfunc* 함수와 일관되도록 만들어야 합니다. *tp_call*을 *PyVectorcall_Call()*로 설정하면 됩니다:

버전 3.8에서 변경: 버전 3.8 이전에는, 이 슬롯의 이름이 tp_print였습니다. 파이썬 2.x에서는, 파일로 인쇄하는 데 사용되었습니다. 파이썬 3.0에서 3.7까지는, 사용되지 않았습니다.

버전 3.12에서 변경: Before version 3.12, it was not recommended for *mutable heap types* to implement the vectorcall protocol. When a user sets __call__ in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function. Since 3.12, setting __call__ will disable vectorcall optimization by clearing the *Py_TPFLAGS_HAVE_VECTORCALL* flag.

계승:

This field is always inherited. However, the *Py_TPFLAGS_HAVE_VECTORCALL* flag is not always inherited. If it' s not set, then the subclass won' t use *vectorcall*, except when *PyVectorcall_Call()* is explicitly called.

getattrfunc *PyTypeObject*.**tp_getattr**

get-attribute-string 함수에 대한 선택적 포인터.

이 필드는 폐지되었습니다. 정의될 때, *tp_getattro* 함수와 같게 작동하지만, 어트리뷰트 이름을 제공하기 위해 파이썬 문자열 객체 대신 C 문자열을 받아들이는 함수를 가리켜야 합니다.

계승:

Group: *tp_getattr*, *tp_getattro*

이 필드는 *tp_getattro*와 함께 서브 형에 의해 상속됩니다: 서브 형은 서브 형의 *tp_getattr*과 *tp_getattro*가 모두 NULL일 때 베이스형에서 *tp_getattr*과 *tp_getattro*를 모두 상속합니다.

setattrfunc *PyTypeObject*.**tp_setattr**

어트리뷰트 설정과 삭제를 위한 함수에 대한 선택적 포인터.

이 필드는 폐지되었습니다. 정의될 때, *tp_setattro* 함수와 같게 작동하지만, 어트리뷰트 이름을 제공하기 위해 파이썬 문자열 객체 대신 C 문자열을 받아들이는 함수를 가리켜야 합니다.

계승:

Group: *tp_setattr*, *tp_setattro*

이 필드는 *tp_setattro*와 함께 서브 형에 의해 상속됩니다. 서브 형은 서브 형의 *tp_setattr*과 *tp_setattro*가 모두 NULL일 때 베이스형에서 *tp_setattr*과 *tp_setattro*를 모두 상속합니다.

*PyAsyncMethods** *PyTypeObject*.**tp_as_async**

C 수준에서 어웨이터블과 비동기 이터레이터 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 자세한 내용은 비동기 객체 구조체를 참조하십시오.

Added in version 3.5: 이전에는 tp_compare와 tp_reserved라고 했습니다.

계승:

tp_as_async 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

reprfunc *PyTypeObject*.**tp_repr**

내장 함수 repr()을 구현하는 함수에 대한 선택적 포인터.

서명은 *PyObject_Repr()*과 같습니다:

```
PyObject *tp_repr(PyObject *self);
```

함수는 문자열이나 유니코드 객체를 반환해야 합니다. 이상적으로, 이 함수는 eval()에 전달될 때 적합한 환경이 주어지면 같은 값을 가진 객체를 반환하는 문자열을 반환해야 합니다. 이것이 가능하지 않으면, '<'로 시작하고 '>'로 끝나는 문자열을 반환해야 하는데, 이 문자열에서 객체의 형과 값을 모두 추론할 수 있어야 합니다.

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

이 필드를 설정하지 않으면, `<%s object at %p>` 형식의 문자열이 반환됩니다. 여기서 `%s`는 형 이름으로, `%p`는 객체의 메모리 주소로 치환됩니다.

PyNumberMethods ******PyTypeObject*`.tp_as_number`

숫자 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 숫자 객체 구조체에서 설명합니다.

계승:

tp_as_number 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

PySequenceMethods ******PyTypeObject*`.tp_as_sequence`

시퀀스 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 시퀀스 객체 구조체에서 설명합니다.

계승:

tp_as_sequence 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

PyMappingMethods ******PyTypeObject*`.tp_as_mapping`

매핑 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 매핑 객체 구조체에서 설명합니다.

계승:

tp_as_mapping 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

hashfunc *PyTypeObject*`.tp_hash`

내장 함수 `hash()`를 구현하는 함수에 대한 선택적 포인터.

서명은 *PyObject_Hash()*와 같습니다:

```
Py_hash_t tp_hash(PyObject *);
```

`-1` 값은 정상적인 반환 값으로 반환되지 않아야 합니다; 해시값을 계산하는 동안 에러가 발생하면 함수는 예외를 설정하고 `-1`을 반환해야 합니다.

When this field is not set (*and* *tp_richcompare* is not set), an attempt to take the hash of the object raises `TypeError`. This is the same as setting it to *PyObject_HashNotImplemented()*.

이 필드는 부모 형에서 해시 메서드의 상속을 차단하기 위해 *PyObject_HashNotImplemented()*로 명시적으로 설정할 수 있습니다. 이것은 파이썬 수준에서의 `__hash__ = None`과 동등한 것으로 해석되어, `isinstance(o, collections.Hashable)`이 `False`를 올바르게 반환하게 합니다. 반대의 경우도 마찬가지입니다 - 파이썬 수준의 클래스에서 `__hash__ = None`을 설정하면 `tp_hash` 슬롯이 *PyObject_HashNotImplemented()*로 설정됩니다.

계승:

Group: *tp_hash*, *tp_richcompare*

이 필드는 *tp_richcompare*와 함께 서브 형에 의해 상속됩니다: 서브 형의 *tp_richcompare*와 *tp_hash*가 모두 `NULL`일 때, 서브 형은 *tp_richcompare*와 *tp_hash*를 모두 상속합니다.

기본값:

PyBaseObject_Type uses *PyObject_GenericHash()*.

ternaryfunc *PyTypeObject*`.tp_call`

객체 호출을 구현하는 함수에 대한 선택적 포인터. 객체가 콜러블이 아니면 `NULL`이어야 합니다. 서명은 *PyObject_Call()*과 같습니다:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

계승:

이 필드는 서브 형으로 상속됩니다.

reprfunc *PyTypeObject*.**tp_str**

내장 연산 `str()`을 구현하는 함수에 대한 선택적 포인터. (`str`는 이제 형이며, `str()`은 그 형의 생성자를 호출함에 유의하십시오. 이 생성자는 *PyObject_Str()*를 호출하여 실제 작업을 수행하고, *PyObject_Str()*은 이 처리기를 호출합니다.)

서명은 *PyObject_Str()*과 같습니다:

```
PyObject *tp_str(PyObject *self);
```

함수는 문자열이나 유니코드 객체를 반환해야 합니다. 다른 것 중에서도, `print()` 함수에 의해 사용될 표현이기 때문에, 객체의 "친숙한" 문자열 표현이어야 합니다.

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

이 필드를 설정하지 않으면, 문자열 표현을 반환하기 위해 *PyObject_Repr()*이 호출됩니다.

getattrofunc *PyTypeObject*.**tp_getattro**

어트리뷰트 읽기(get-attribute) 함수에 대한 선택적 포인터.

서명은 *PyObject_GetAttr()*과 같습니다:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

일반적으로 이 필드를 *PyObject_GenericGetAttr()*로 설정하는 것이 편리합니다, 객체 어트리뷰트를 찾는 일반적인 방법을 구현합니다.

계승:

Group: *tp_getattr*, *tp_getattro*

이 필드는 *tp_getattr*과 함께 서브 형에 의해 상속됩니다: 서브 형의 *tp_getattr*과 *tp_getattro*가 모두 `NULL`일 때 서브 형은 베이스형에서 *tp_getattr*과 *tp_getattro*를 모두 상속합니다.

기본값:

PyBaseObject_Type uses *PyObject_GenericGetAttr()*.

setattrofunc *PyTypeObject*.**tp_setattro**

어트리뷰트 설정과 삭제를 위한 함수에 대한 선택적 포인터.

서명은 *PyObject_SetAttr()*과 같습니다:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

또한, *value*를 `NULL`로 설정하여 어트리뷰트를 삭제하는 것을 반드시 지원해야 합니다. 일반적으로 이 필드를 *PyObject_GenericSetAttr()*로 설정하는 것이 편리합니다, 객체 어트리뷰트를 설정하는 일반적인 방법을 구현합니다.

계승:

Group: *tp_setattr*, *tp_setattro*

이 필드는 *tp_setattr*과 함께 서브 형에 의해 상속됩니다: 서브 형의 *tp_setattr*과 *tp_setattro*가 모두 `NULL`일 때, 서브 형은 베이스형에서 *tp_setattr*과 *tp_setattro*를 모두 상속합니다.

기본값:

PyBaseObject_Type uses *PyObject_GenericSetAttr()*.

PyBufferProcs **PyTypeObject.tp_as_buffer*

버퍼 인터페이스를 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 버퍼 객체 구조체에서 설명합니다.

계승:

tp_as_buffer 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

unsigned long *PyTypeObject*.**tp_flags**

이 필드는 다양한 플래그의 비트 마스크입니다. 일부 플래그는 특정 상황에 대한 변형 의미론을 나타냅니다; 다른 것들은 역사적으로 항상 존재하지는 않았던 형 객체(또는 *tp_as_number*, *tp_as_sequence*, *tp_as_mapping* 및 *tp_as_buffer*를 통해 참조되는 확장 구조체)의 특정 필드가 유효함을 나타내는 데 사용됩니다; 이러한 플래그 비트가 없으면, 이것이 보호하는 형 필드에 액세스하지 말아야 하며 대신 0이나 `NULL` 값을 갖는 것으로 간주해야 합니다.

계승:

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type' s value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have `NULL` values.

기본값:

PyBaseObject_Type uses `Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE`.

비트 마스크:

다음 비트 마스크가 현재 정의되어 있습니다; 이들은 `|` 연산자로 함께 OR 하여 *tp_flags* 필드의 값을 형성할 수 있습니다. 매크로 *PyType_HasFeature()*는 형과 플래그 값 *tp*와 *f*를 취하고 `tp->tp_flags & f`가 0이 아닌지 확인합니다.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using *PyType_FromSpec()*. In this case, the *ob_type* field of its instances is considered a reference to the type, and the type object is INCREF' ed when a new instance is created, and DECREF' ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance' s ob_type gets INCREF' ed or DECREF' ed). Heap types should also *support garbage collection* as they can form a reference cycle with their own module object.

계승:

???

Py_TPFLAGS_BASETYPE

이 비트는 형을 다른 형의 베이스형으로 사용할 수 있을 때 설정됩니다. 이 비트가 설정되지 않으면 이 형으로 서브 형을 만들 수 없습니다 (Java의 "final" 클래스와 유사합니다).

계승:

???

Py_TPFLAGS_READY

이 비트는 *PyType_Ready()*에 의해 형 객체가 완전히 초기화될 때 설정됩니다.

계승:

???

Py_TPFLAGS_READYING

이 비트는 *PyType_Ready()*가 형 객체를 초기화하는 동안 설정됩니다.

계승:

???

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, memory for new instances (see *tp_alloc*) must be allocated using *PyObject_GC_New* or *PyType_GenericAlloc()* and deallocated (see *tp_free*) using *PyObject_GC_Del()*. More information in section 순환 가비지 수집 지원.

계승:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have `NULL` values.

Py_TPFLAGS_DEFAULT

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`.

계승:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

이 비트는 객체가 연결되지 않은 메서드(unbound method)처럼 동작함을 나타냅니다.

이 플래그가 `type(meth)` 에 설정되면:

- `meth.__get__(obj, cls)(*args, **kwds)`(`obj` 가 None 이 아닐 때)는 `meth(obj, *args, **kwds)` 와 동등해야 합니다.
- `meth.__get__(None, cls)(*args, **kwds)` 는 `meth(*args, **kwds)` 와 동등해야 합니다.

이 플래그는 `obj.meth()` 와 같은 일반적인 메서드 호출에 대한 최적화를 가능하게 합니다: `obj.meth`에 대한 임시 "연결된 메서드(bound method)" 객체를 만들지 않습니다.

Added in version 3.8.

계승:

This flag is never inherited by types without the *Py_TPFLAGS_IMMUTABLETYPE* flag set. For extension types, it is inherited whenever *tp_descr_get* is inherited.

Py_TPFLAGS_MANAGED_DICT

This bit indicates that instances of the class have a `__dict__` attribute, and that the space for the dictionary is managed by the VM.

If this flag is set, *Py_TPFLAGS_HAVE_GC* should also be set.

The type traverse function must call *PyObject_VisitManagedDict()* and its clear function must call *PyObject_ClearManagedDict()*.

Added in version 3.12.

계승:

This flag is inherited unless the *tp_dictoffset* field is set in a superclass.

Py_TPFLAGS_MANAGED_WEAKREF

This bit indicates that instances of the class should be weakly referenceable.

Added in version 3.12.

계승:

This flag is inherited unless the *tp_weaklistoffset* field is set in a superclass.

Py_TPFLAGS_ITEMS_AT_END

Only usable with variable-size types, i.e. ones with non-zero *tp_itemsize*.

Indicates that the variable-sized portion of an instance of this type is at the end of the instance' s memory area, at an offset of `Py_TYPE(obj)->tp_basicsize` (which may be different in each subclass).

When setting this flag, be sure that all superclasses either use this memory layout, or are not variable-sized. Python does not check this.

Added in version 3.12.

계승:

This flag is inherited.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS

이 플래그는 *PyLong_Check()* 와 같은 함수에서 형이 내장형의 서브 클래스인지 신속하게 판별하는 데 사용됩니다; 이러한 특정 검사는 *PyObject_IsInstance()*와 같은 일반 검사보다 빠릅니다. 내장에서 상속된 사용자 정의 형은 *tp_flags*를 적절하게 설정해야 합니다, 그렇지 않으면 그러한 형과 상호 작용하는 코드가 사용되는 검사의 유형에 따라 다르게 작동합니다.

Py_TPFLAGS_HAVE_FINALIZE

이 비트는 *tp_finalize* 슬롯이 형 구조체에 있을 때 설정됩니다.

Added in version 3.4.

버전 3.8부터 폐지됨: 인터프리터는 *tp_finalize* 슬롯이 항상 형 구조체에 있다고 가정하기 때문에, 이 플래그는 더는 필요하지 않습니다.

Py_TPFLAGS_HAVE_VECTORCALL

이 비트는 클래스가 벡터콜 프로토콜을 구현할 때 설정됩니다. 자세한 내용은 *tp_vectorcall_offset*을 참조하십시오.

계승:

This bit is inherited if *tp_call* is also inherited.

Added in version 3.9.

버전 3.12에서 변경: This flag is now removed from a class when the class' s `__call__()` method is reassigned.

This flag can now be inherited by mutable classes.

Py_TPFLAGS_IMMUTABLETYPE

This bit is set for type objects that are immutable: type attributes cannot be set nor deleted.

PyType_Ready() automatically applies this flag to *static types*.

계승:

This flag is not inherited.

Added in version 3.10.

Py_TPFLAGS_DISALLOW_INSTANTIATION

Disallow creating instances of the type: set *tp_new* to NULL and don' t create the `__new__` key in the type dictionary.

The flag must be set before creating the type, not after. For example, it must be set before *PyType_Ready()* is called on the type.

The flag is set automatically on *static types* if *tp_base* is NULL or `&PyBaseObject_Type` and *tp_new* is NULL.

계승:

This flag is not inherited. However, subclasses will not be instantiable unless they provide a non-NULL *tp_new* (which is only possible via the C API).

> **참고**
>
> To disallow instantiating a class directly but allow instantiating its subclasses (e.g. for an *abstract base class*), do not use this flag. Instead, make *tp_new* only succeed for subclasses.

Added in version 3.10.

Py_TPFLAGS_MAPPING

This bit indicates that instances of the class may match mapping patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Mapping`, and unset when registering `collections.abc.Sequence`.

> **참고**
>
> *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

계승:

This flag is inherited by types that do not already set *Py_TPFLAGS_SEQUENCE*.

> **더 보기**
>
> **PEP 634** – Structural Pattern Matching: Specification

Added in version 3.10.

Py_TPFLAGS_SEQUENCE

This bit indicates that instances of the class may match sequence patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Sequence`, and unset when registering `collections.abc.Mapping`.

> **참고**
>
> *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

계승:

This flag is inherited by types that do not already set *Py_TPFLAGS_MAPPING*.

> **더 보기**
>
> **PEP 634** – Structural Pattern Matching: Specification

Added in version 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

Internal. Do not set or unset this flag. To indicate that a class has changed call *PyType_Modified()*

> **경고**
>
> This flag is present in header files, but is not be used. It will be removed in a future version of CPython

const char **PyTypeObject*.**tp_doc**

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the `__doc__` attribute on the type and instances of the type.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

traverseproc *PyTypeObject*.**tp_traverse**

An optional pointer to a traversal function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

파이썬의 가비지 수집 체계에 대한 자세한 정보는 섹션 순환 가비지 수집 지원에서 찾을 수 있습니다.

The *tp_traverse* pointer is used by the garbage collector to detect reference cycles. A typical implementation of a *tp_traverse* function simply calls *Py_VISIT()* on each of the instance' s members that are Python objects that the instance owns. For example, this is function `local_traverse()` from the `_thread` extension module:

```
static int
local_traverse(PyObject *op, visitproc visit, void *arg)
{
    localobject *self = (localobject *) op;
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

*Py_VISIT()*는 참조 순환에 참여할 수 있는 멤버에 대해서만 호출됨에 유의하십시오. `self->key` 멤버도 있지만, `NULL`이나 파이썬 문자열만 가능해서 참조 순환의 일부가 될 수 없습니다.

반면에, 멤버가 사이클의 일부가 될 수 없다는 것을 알고 있더라도, 디버깅 지원을 위해 `gc` 모듈의 `get_referents()` 함수가 그것을 포함하도록 어쨌거나 방문하고 싶을 수 있습니다.

Heap types (*Py_TPFLAGS_HEAPTYPE*) must visit their type with:

```
Py_VISIT(Py_TYPE(self));
```

It is only needed since Python 3.9. To support Python 3.8 and older, this line must be conditional:

```
#if PY_VERSION_HEX >= 0x03090000
    Py_VISIT(Py_TYPE(self));
#endif
```

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_VisitManagedDict()* like this:

```
PyObject_VisitManagedDict((PyObject*)self, visit, arg);
```

경고

When implementing *tp_traverse*, only the members that the instance *owns* (by having *strong references* to them) must be visited. For instance, if an object supports weak references via the *tp_weaklist* slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Note that *Py_VISIT()* requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don' t name them just anything.

Instances of *heap-allocated types* hold a reference to their type. Their traversal function must therefore either visit *Py_TYPE(self)*, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

참고

The *tp_traverse* function can be called from any thread.

버전 3.9에서 변경: 힙 할당 형은 `tp_traverse`에서 `Py_TYPE(self)`를 방문할 것으로 기대됩니다. 이전 버전의 파이썬에서는, 버그 40217로 인해, 이렇게 하면 서브 클래스에서 충돌이 발생할 수 있습니다.

계승:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_clear* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

inquiry *PyTypeObject*.**tp_clear**

An optional pointer to a clear function. The signature is:

```
int tp_clear(PyObject *);
```

The purpose of this function is to break reference cycles that are causing a *cyclic isolate* so that the objects can be safely destroyed. A cleared object is a partially destroyed object; the object is not obligated to satisfy design invariants held during normal use.

`tp_clear` does not need to delete references to objects that can' t participate in reference cycles, such as Python strings or Python integers. However, it may be convenient to clear all references, and write the type' s *tp_dealloc* function to invoke `tp_clear` to avoid code duplication. (Beware that `tp_clear` might have already been called. Prefer calling idempotent functions like *Py_CLEAR()*.)

Any non-trivial cleanup should be performed in *tp_finalize* instead of `tp_clear`.

참고

If `tp_clear` fails to break a reference cycle then the objects in the *cyclic isolate* may remain indefinitely uncollectable ("leak"). See `gc.garbage`.

> **참고**
>
> Referents (direct and indirect) might have already been cleared; they are not guaranteed to be in a consistent state.

> **참고**
>
> The *tp_clear* function can be called from any thread.

> **참고**
>
> An object is not guaranteed to be automatically cleared before its destructor (*tp_dealloc*) is called.

This function differs from the destructor (*tp_dealloc*) in the following ways:

- The purpose of clearing an object is to remove references to other objects that might participate in a reference cycle. The purpose of the destructor, on the other hand, is a superset: it must release *all* resources it owns, including references to objects that cannot participate in a reference cycle (e.g., integers) as well as the object' s own memory (by calling *tp_free*).
- When `tp_clear` is called, other objects might still hold references to the object being cleared. Because of this, `tp_clear` must not deallocate the object' s own memory (*tp_free*). The destructor, on the other hand, is only called when no (strong) references exist, and as such, must safely destroy the object itself by deallocating it.
- `tp_clear` might never be automatically called. An object' s destructor, on the other hand, will be automatically called some time after the object becomes unreachable (i.e., either there are no references to the object or the object is a member of a *cyclic isolate*).

No guarantees are made about when, if, or how often Python automatically clears an object, except:

- Python will not automatically clear an object if it is reachable, i.e., there is a reference to it and it is not a member of a *cyclic isolate*.
- Python will not automatically clear an object if it has not been automatically finalized (see *tp_finalize*). (If the finalizer resurrected the object, the object may or may not be automatically finalized again before it is cleared.)
- If an object is a member of a *cyclic isolate*, Python will not automatically clear it if any member of the cyclic isolate has not yet been automatically finalized (*tp_finalize*).
- Python will not destroy an object until after any automatic calls to its `tp_clear` function have returned. This ensures that the act of breaking a reference cycle does not invalidate the `self` pointer while `tp_clear` is still executing.
- Python will not automatically call `tp_clear` multiple times concurrently.

CPython currently only automatically clears objects as needed to break reference cycles in a *cyclic isolate*, but future versions might clear objects regularly before their destruction.

Taken together, all *tp_clear* functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a *tp_clear* function. For example, the tuple type does not implement a *tp_clear* function, because it' s possible to prove that no reference cycle can be composed entirely of tuples. Therefore the *tp_clear* functions of other types are responsible for breaking any cycle containing a tuple. This isn' t immediately obvious, and there' s rarely a good reason to avoid implementing *tp_clear*.

*tp_clear*의 구현은 다음 예제와 같이 파이썬 객체일 수 있는 자신의 멤버에 대한 인스턴스의 참조를 삭제하고 해당 멤버에 대한 포인터를 `NULL`로 설정해야 합니다:

```
static int
local_clear(PyObject *op)
{
    localobject *self = (localobject *) op;
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate: the reference to the contained object must not be released (via *Py_DECREF()*) until after the pointer to the contained object is set to NULL. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it' s possible for such code to reference *self* again, it' s important that the pointer to the contained object be NULL at that time, so that *self* knows the contained object can no longer be used. The *Py_CLEAR()* macro performs the operations in a safe order.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_ClearManagedDict()* like this:

```
PyObject_ClearManagedDict((PyObject*)self);
```

파이썬의 가비지 수집 체계에 대한 자세한 정보는 섹션 순환 가비지 수집 지원에서 찾을 수 있습니다.

계승:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_traverse* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

더 보기

Object Life Cycle for details about how this slot relates to other slots.

richcmpfunc *PyTypeObject*.**tp_richcompare**

풍부한 비교 함수(rich comparison function)에 대한 선택적 포인터. 서명은 다음과 같습니다:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

첫 번째 매개 변수는 *PyTypeObject*에 의해 정의된 형의 인스턴스임이 보장됩니다.

이 함수는 비교 결과(일반적으로 Py_True나 Py_False)를 반환해야 합니다. 비교가 정의되어 있지 않으면, Py_NotImplemented를 반환하고, 다른 에러가 발생하면 NULL을 반환하고 예외 조건을 설정해야 합니다.

다음 상수는 *tp_richcompare*와 *PyObject_RichCompare()*의 세 번째 인자로 사용되도록 정의됩니다:

상수	비교
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

풍부한 비교 함수를 쉽게 작성할 수 있도록 다음 매크로가 정의됩니다:

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

비교 결과에 따라, 함수에서 `Py_True`나 `Py_False`를 반환합니다. VAL_A와 VAL_B는 C 비교 연산자로 순서를 정할 수 있어야 합니다 (예를 들어, C int나 float일 수 있습니다). 세 번째 인자는 *PyObject_RichCompare()*에서처럼 요청된 연산을 지정합니다.

The returned value is a new *strong reference*.

에러가 발생하면, 예외를 설정하고 함수에서 `NULL`을 반환합니다.

Added in version 3.7.

계승:

Group: *tp_hash*, *tp_richcompare*

이 필드는 *tp_hash*와 함께 서브 형에 의해 상속됩니다. 서브 형의 *tp_richcompare*와 *tp_hash*가 모두 `NULL`이면 서브 형은 *tp_richcompare*와 *tp_hash*를 상속합니다.

기본값:

PyBaseObject_Type provides a *tp_richcompare* implementation, which may be inherited. However, if only *tp_hash* is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t *PyTypeObject*.**tp_weaklistoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_WEAKREF* should be used instead, if at all possible.

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by *PyObject_ClearWeakRefs()* and the `PyWeakref_*` functions. The instance structure needs to include a field of type *PyObject*`*` which is initialized to `NULL`.

이 필드를 *tp_weaklist*와 혼동하지 마십시오; 그것은 형 객체 자체에 대한 약한 참조의 리스트 헤드입니다.

It is an error to set both the *Py_TPFLAGS_MANAGED_WEAKREF* bit and *tp_weaklistoffset*.

계승:

이 필드는 서브 형에 의해 상속되지만, 아래 나열된 규칙을 참조하십시오. 서브 형이 이 오프셋을 재정의할 수 있습니다; 이는 서브 형이 베이스형과 다른 약한 참조 리스트 헤드를 사용함을 의미합니다. 리스트 헤드는 항상 *tp_weaklistoffset*을 통해 발견되므로, 문제가 되지 않습니다.

기본값:

If the *Py_TPFLAGS_MANAGED_WEAKREF* bit is set in the *tp_flags* field, then *tp_weaklistoffset* will be set to a negative value, to indicate that it is unsafe to use this field.

getiterfunc *PyTypeObject*.**tp_iter**

An optional pointer to a function that returns an *iterator* for the object. Its presence normally signals that the instances of this type are *iterable* (although sequences may be iterable without this function).

이 함수는 *PyObject_GetIter()*와 같은 서명을 갖습니다:

```
PyObject *tp_iter(PyObject *self);
```

계승:

이 필드는 서브 형으로 상속됩니다.

iternextfunc *PyTypeObject*.**tp_iternext**

An optional pointer to a function that returns the next item in an *iterator*. The signature is:

```
PyObject *tp_iternext(PyObject *self);
```

이터레이터가 소진되면 `NULL`을 반환해야 합니다; `StopIteration` 예외가 설정될 수도, 그렇지 않을 수도 있습니다. 다른 에러가 발생하면, 역시 `NULL`을 반환해야 합니다. 그 존재는 이 형의 인스턴스가 이터레이터라는 신호입니다.

이터레이터 형은 *tp_iter* 함수도 정의해야 하며, 해당 함수는 (새 이터레이터 인스턴스가 아닌) 이터레이터 인스턴스 자체를 반환해야 합니다.

이 함수는 *PyIter_Next()*와 같은 서명을 갖습니다.

계승:

이 필드는 서브 형으로 상속됩니다.

struct *PyMethodDef* **PyTypeObject*.**tp_methods**

이 형의 일반 메서드를 선언하는 *PyMethodDef* 구조체의 정적 `NULL`-종료 배열에 대한 선택적 포인터.

배열의 항목마다, 메서드 디스크립터를 포함하는 형의 딕셔너리(아래 *tp_dict*를 참조하십시오)에 항목이 추가됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다 (메서드는 다른 메커니즘을 통해 상속됩니다).

struct *PyMemberDef* **PyTypeObject*.**tp_members**

이 형의 인스턴스의 일반 데이터 멤버(필드나 슬롯)를 선언하는 *PyMemberDef* 구조체의 정적 `NULL`-종료 배열에 대한 선택적 포인터.

배열의 항목마다, 멤버 디스크립터를 포함하는 형의 딕셔너리(아래 *tp_dict*를 참조하십시오)에 항목이 추가됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다 (멤버는 다른 메커니즘을 통해 상속됩니다).

struct *PyGetSetDef* **PyTypeObject*.**tp_getset**

이 형의 인스턴스의 계산된 어트리뷰트를 선언하는 *PyGetSetDef* 구조체의 정적 `NULL`-종료 배열에 대한 선택적 포인터.

배열의 항목마다, getset 디스크립터를 포함하는 형의 딕셔너리(아래 *tp_dict*를 참조하십시오)에 항목이 추가됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다 (계산된 어트리뷰트는 다른 메커니즘을 통해 상속됩니다).

PyTypeObject **PyTypeObject.tp_base**

형 속성이 상속되는 베이스형에 대한 선택적 포인터. 이 수준에서는, 단일 상속만 지원됩니다; 다중 상속은 메타 형을 호출하여 형 객체를 동적으로 작성해야 합니다.

> **참고**
>
> 슬롯 초기화에는 전역 초기화 규칙이 적용됩니다. C99에서는 초기화자가 "주소 상수(address constants)"여야 합니다. 포인터로 묵시적으로 변환되는 *PyType_GenericNew()*와 같은 함수 지정자는 유효한 C99 주소 상수입니다.
>
> However, the unary '&' operator applied to a non-static variable like *PyBaseObject_Type* is not required to produce an address constant. Compilers may support this (gcc does), MSVC does not. Both compilers are strictly standard conforming in this particular behavior.
>
> 결과적으로, *tp_base*는 확장 모듈의 초기화 함수에서 설정되어야 합니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다 (명백히).

기본값:

이 필드의 기본값은 `&PyBaseObject_Type`입니다 (파이썬 프로그래머에게는 `object` 형으로 알려져 있습니다).

PyObject **PyTypeObject.tp_dict**

형의 딕셔너리는 *PyType_Ready()*에 의해 여기에 저장됩니다.

This field should normally be initialized to `NULL` before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once *PyType_Ready()* has initialized the type, extra attributes for the type may be added to this dictionary only if they don' t correspond to overloaded operations (like `__add__()`). Once initialization for the type has finished, this field should be treated as read-only.

Some types may not store their dictionary in this slot. Use *PyType_GetDict()* to retrieve the dictionary for an arbitrary type.

버전 3.12에서 변경: Internals detail: For static builtin types, this is always `NULL`. Instead, the dict for such types is stored on `PyInterpreterState`. Use *PyType_GetDict()* to get the dict for an arbitrary type.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다 (여기에 정의된 어트리뷰트는 다른 메커니즘을 통해 상속됩니다).

기본값:

이 필드가 `NULL`이면, *PyType_Ready()*는 새 딕셔너리를 할당합니다.

> **경고**
>
> *PyDict_SetItem()*을 사용하거나 다른 식으로 딕셔너리 C-API로 *tp_dict*를 수정하는 것은 안전하지 않습니다.

descrgetfunc *PyTypeObject*.**tp_descr_get**

"디스크립터 get" 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

계승:

이 필드는 서브 형으로 상속됩니다.

descrsetfunc *PyTypeObject*.**tp_descr_set**

디스크립터 값을 설정하고 삭제하기 위한 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

value 인자는 값을 삭제하기 위해 NULL로 설정됩니다.

계승:

이 필드는 서브 형으로 상속됩니다.

Py_ssize_t *PyTypeObject*.**tp_dictoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_DICT* should be used instead, if at all possible.

이 형의 인스턴스에 인스턴스 변수를 포함하는 딕셔너리가 있으면, 이 필드는 0이 아니며 인스턴스 변수 딕셔너리 형의 인스턴스에서의 오프셋을 포함합니다; 이 오프셋은 *PyObject_GenericGetAttr()*에서 사용됩니다.

이 필드를 *tp_dict*와 혼동하지 마십시오; 그것은 형 객체 자체의 어트리뷰트에 대한 딕셔너리입니다.

The value specifies the offset of the dictionary from the start of the instance structure.

The *tp_dictoffset* should be regarded as write-only. To get the pointer to the dictionary call *PyObject_GenericGetDict()*. Calling *PyObject_GenericGetDict()* may need to allocate memory for the dictionary, so it is may be more efficient to call *PyObject_GetAttr()* when accessing an attribute on the object.

It is an error to set both the *Py_TPFLAGS_MANAGED_DICT* bit and *tp_dictoffset*.

계승:

This field is inherited by subtypes. A subtype should not override this offset; doing so could be unsafe, if C code tries to access the dictionary at the previous offset. To properly support inheritance, use *Py_TPFLAGS_MANAGED_DICT*.

기본값:

This slot has no default. For *static types*, if the field is NULL then no __dict__ gets created for instances.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, then *tp_dictoffset* will be set to -1, to indicate that it is unsafe to use this field.

initproc *PyTypeObject*.**tp_init**

인스턴스 초기화 함수에 대한 선택적 포인터.

This function corresponds to the __init__() method of classes. Like __init__(), it is possible to create an instance without calling __init__(), and it is possible to reinitialize an instance by calling its __init__() method again.

함수 서명은 다음과 같습니다:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

The self argument is the instance to be initialized; the *args* and *kwds* arguments represent positional and keyword arguments of the call to __init__().

NULL이 아닐 때, *tp_init* 함수는 형을 호출하여 인스턴스를 정상적으로 만들 때, 형의 *tp_new* 함수가 형의 인스턴스를 반환한 후 호출됩니다. *tp_new* 함수가 원래 형의 서브 형이 아닌 다른 형의 인스턴스를 반환하면, 아무런 *tp_init* 함수도 호출되지 않습니다; *tp_new*가 원래 형의 서브 형 인스턴스를 반환하면, 서브 형의 *tp_init*가 호출됩니다.

성공하면 0을 반환하고, 에러 시에는 -1을 반환하고 예외를 설정합니다.

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

For *static types* this field does not have a default.

allocfunc *PyTypeObject*.**tp_alloc**

인스턴스 할당 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

계승:

Static subtypes inherit this slot, which will be *PyType_GenericAlloc()* if inherited from `object`.

Heap subtypes do not inherit this slot.

기본값:

For heap subtypes, this field is always set to *PyType_GenericAlloc()*.

For static subtypes, this slot is inherited (see above).

newfunc *PyTypeObject*.**tp_new**

인스턴스 생성 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

subtype 인자는 만들어지고 있는 객체의 형입니다; *args*와 *kwds* 인자는 형 호출의 위치와 키워드 인자를 나타냅니다. *subtype*이 *tp_new* 함수가 호출되는 형과 같을 필요는 없음에 유의하십시오; 이 형의 서브 형일 수 있습니다 (하지만 관련이 없는 형은 아닙니다).

tp_new 함수는 객체에 공간을 할당하기 위해 `subtype->tp_alloc(subtype, nitems)`를 호출해야 하고, 그런 다음 꼭 필요한 만큼만 추가 초기화를 수행해야 합니다. 안전하게 무시하거나 반복할 수 있는 초기화는 *tp_init* 처리기에 배치해야 합니다. 간단한 규칙은, 불변 형의 경우 모든 초기화가 *tp_new*에서 수행되어야 하고, 가변형의 경우 대부분 초기화는 *tp_init*로 미뤄져야 합니다.

Set the *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag to disallow creating instances of the type in Python.

계승:

This field is inherited by subtypes, except it is not inherited by *static types* whose *tp_base* is `NULL` or `&PyBaseObject_Type`.

기본값:

For *static types* this field has no default. This means if the slot is defined as `NULL`, the type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

freefunc *PyTypeObject*.**tp_free**

인스턴스 할당 해제 함수에 대한 선택적 포인터. 서명은 다음과 같습니다:

```
void tp_free(void *self);
```

This function must free the memory allocated by *tp_alloc*.

계승:

Static subtypes inherit this slot, which will be *PyObject_Free()* if inherited from `object`. Exception: If the type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*) and it would inherit *PyObject_Free()*, then this slot is not inherited but instead defaults to *PyObject_GC_Del()*.

Heap subtypes do not inherit this slot.

기본값:

For *heap subtypes*, this slot defaults to a deallocator suitable to match `PyType_GenericAlloc()` and the value of the `Py_TPFLAGS_HAVE_GC` flag.

For static subtypes, this slot is inherited (see above).

inquiry `PyTypeObject`**.tp_is_gc**

가비지 수집기에서 호출되는 함수에 대한 선택적 포인터.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object' s type' s `tp_flags` field, and check the `Py_TPFLAGS_HAVE_GC` flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is:

```
int tp_is_gc(PyObject *self);
```

(The only example of this are types themselves. The metatype, `PyType_Type`, defines this function to distinguish between statically and *dynamically allocated types*.)

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

This slot has no default. If this field is `NULL`, `Py_TPFLAGS_HAVE_GC` is used as the functional equivalent.

PyObject *`PyTypeObject`**.tp_bases**

베이스형의 튜플.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is `initialized`.

For dynamically created classes, the `Py_tp_bases` `slot` can be used instead of the *bases* argument of `PyType_FromSpecWithBases()`. The argument form is preferred.

경고

Multiple inheritance does not work well for statically defined types. If you set `tp_bases` to a tuple, Python will not raise an error, but some slots will only be inherited from the first base.

계승:

이 필드는 상속되지 않습니다.

PyObject *`PyTypeObject`**.tp_mro**

형 자체에서 시작하여 `object`로 끝나는 확장된 베이스형 집합을 포함하는 튜플.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is `initialized`.

계승:

이 필드는 상속되지 않습니다; `PyType_Ready()`에 의해 새로 계산됩니다.

PyObject *`PyTypeObject`**.tp_cache**

사용되지 않습니다. 내부 전용.

계승:

이 필드는 상속되지 않습니다.

void *`PyTypeObject`**.tp_subclasses**

A collection of subclasses. Internal use only. May be an invalid pointer.

To get a list of subclasses, call the Python method `__subclasses__()`.

버전 3.12에서 변경: For some types, this field does not hold a valid `PyObject*`. The type was changed to `void*` to indicate this.

계승:

이 필드는 상속되지 않습니다.

PyObject ******PyTypeObject*.**tp_weaklist**

이 형 객체에 대한 약한 참조를 위한 약한 참조 리스트 헤드. 상속되지 않습니다. 내부 전용.

버전 3.12에서 변경: Internals detail: For the static builtin types this is always `NULL`, even if weakrefs are added. Instead, the weakrefs for each are stored on `PyInterpreterState`. Use the public C-API or the internal `_PyObject_GET_WEAKREFS_LISTPTR()` macro to avoid the distinction.

계승:

이 필드는 상속되지 않습니다.

destructor *PyTypeObject*.**tp_del**

이 필드는 폐지되었습니다. 대신 *tp_finalize*를 사용하십시오.

unsigned int *PyTypeObject*.**tp_version_tag**

메서드 캐시에 인덱싱하는 데 사용됩니다. 내부 전용.

계승:

이 필드는 상속되지 않습니다.

destructor *PyTypeObject*.**tp_finalize**

An optional pointer to an instance finalization function. This is the C implementation of the `__del__()` special method. Its signature is:

```
void tp_finalize(PyObject *self);
```

The primary purpose of finalization is to perform any non-trivial cleanup that must be performed before the object is destroyed, while the object and any other objects it directly or indirectly references are still in a consistent state. The finalizer is allowed to execute arbitrary Python code.

Before Python automatically finalizes an object, some of the object' s direct or indirect referents might have themselves been automatically finalized. However, none of the referents will have been automatically cleared (*tp_clear*) yet.

Other non-finalized objects might still be using a finalized object, so the finalizer must leave the object in a sane state (e.g., invariants are still met).

참고

After Python automatically finalizes an object, Python might start automatically clearing (*tp_clear*) the object and its referents (direct and indirect). Cleared objects are not guaranteed to be in a consistent state; a finalized object must be able to tolerate cleared referents.

참고

An object is not guaranteed to be automatically finalized before its destructor (*tp_dealloc*) is called. It is recommended to call *PyObject_CallFinalizerFromDealloc()* at the beginning of `tp_dealloc` to guarantee that the object is always finalized before destruction.

참고

The *tp_finalize* function can be called from any thread, although the *GIL* will be held.

> **참고**
>
> The `tp_finalize` function can be called during shutdown, after some global variables have been deleted. See the documentation of the `__del__()` method for details.

When Python finalizes an object, it behaves like the following algorithm:

1. Python might mark the object as *finalized*. Currently, Python always marks objects whose type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*) and never marks other types of objects; this might change in a future version.
2. If the object is not marked as *finalized* and its `tp_finalize` finalizer function is non-`NULL`, the finalizer function is called.
3. If the finalizer function was called and the finalizer made the object reachable (i.e., there is a reference to the object and it is not a member of a *cyclic isolate*), then the finalizer is said to have *resurrected* the object. It is unspecified whether the finalizer can also resurrect the object by adding a new reference to the object that does not make it reachable, i.e., the object is (still) a member of a cyclic isolate.
4. If the finalizer resurrected the object, the object' s pending destruction is canceled and the object' s *finalized* mark might be removed if present. Currently, Python never removes the *finalized* mark; this might change in a future version.

Automatic finalization refers to any finalization performed by Python except via calls to *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()*. No guarantees are made about when, if, or how often an object is automatically finalized, except:

- Python will not automatically finalize an object if it is reachable, i.e., there is a reference to it and it is not a member of a *cyclic isolate*.
- Python will not automatically finalize an object if finalizing it would not mark the object as *finalized*. Currently, this applies to objects whose type does not support garbage collection, i.e., the *Py_TPFLAGS_HAVE_GC* flag is not set. Such objects can still be manually finalized by calling *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()*.
- Python will not automatically finalize any two members of a *cyclic isolate* concurrently.
- Python will not automatically finalize an object after it has automatically cleared (*tp_clear*) the object.
- If an object is a member of a *cyclic isolate*, Python will not automatically finalize it after automatically clearing (see *tp_clear*) any other member.
- Python will automatically finalize every member of a *cyclic isolate* before it automatically clears (see *tp_clear*) any of them.
- If Python is going to automatically clear an object (*tp_clear*), it will automatically finalize the object first.

Python currently only automatically finalizes objects that are members of a *cyclic isolate*, but future versions might finalize objects regularly before their destruction.

To manually finalize an object, do not call this function directly; call *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()* instead.

tp_finalize should leave the current exception status unchanged. The recommended way to write a non-trivial finalizer is to back up the exception at the beginning by calling *PyErr_GetRaisedException()* and restore the exception at the end by calling *PyErr_SetRaisedException()*. If an exception is encountered in the middle of the finalizer, log and clear it with *PyErr_WriteUnraisable()* or *PyErr_FormatUnraisable()*. For example:

```
static void
foo_finalize(PyObject *self)
{
    // Save the current exception, if any.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    PyObject *exc = PyErr_GetRaisedException();

    // ...

    if (do_something_that_might_raise() != success_indicator) {
        PyErr_WriteUnraisable(self);
        goto done;
    }

done:
    // Restore the saved exception.  This silently discards any exception
    // raised above, so be sure to call PyErr_WriteUnraisable first if
    // necessary.
    PyErr_SetRaisedException(exc);
}
```

계승:

이 필드는 서브 형으로 상속됩니다.

Added in version 3.4.

버전 3.8에서 변경: Before version 3.8 it was necessary to set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit in order for this field to be used. This is no longer required.

더 보기

- **PEP 442**: “Safe object finalization”
- *Object Life Cycle* for details about how this slot relates to other slots.
- *PyObject_CallFinalizer()*
- *PyObject_CallFinalizerFromDealloc()*

vectorcallfunc *PyTypeObject*.**tp_vectorcall**

A *vectorcall function* to use for calls of this type object (rather than instances). In other words, `tp_vectorcall` can be used to optimize `type.__call__`, which typically returns a new instance of *type*.

As with any vectorcall function, if `tp_vectorcall` is `NULL`, the *tp_call* protocol (`Py_TYPE(type)->tp_call`) is used instead.

참고

The *vectorcall protocol* requires that the vectorcall function has the same behavior as the corresponding `tp_call`. This means that `type->tp_vectorcall` must match the behavior of `Py_TYPE(type)->tp_call`.

Specifically, if *type* uses the default metaclass, `type->tp_vectorcall` must behave the same as *PyType_Type*`->tp_call`, which:

- calls `type->tp_new`,
- if the result is a subclass of *type*, calls `type->tp_init` on the result of `tp_new`, and
- returns the result of `tp_new`.

Typically, `tp_vectorcall` is overridden to optimize this process for specific *tp_new* and *tp_init*. When doing this for user-subclassable types, note that both can be overridden (using `__new__()` and `__init__()`, respectively).

계승:

이 필드는 상속되지 않습니다.

Added in version 3.9: (필드는 3.8부터 존재하지만 3.9부터 사용됩니다)

unsigned char *PyTypeObject*.**tp_watched**

Internal. Do not use.

Added in version 3.12.

13.4.6 Static Types

전통적으로, C 코드에서 정의된 형은 정적(*static*)입니다. 즉 정적 *PyTypeObject* 구조체는 코드에서 직접 정의되고 *PyType_Ready()*를 사용하여 초기화됩니다.

결과적으로 파이썬에서 정의된 형에 비해 형이 제한됩니다:

- 정적 형은 하나의 베이스로 제한됩니다. 즉, 다중 상속을 사용할 수 없습니다.
- 정적 형 객체(그러나 이들의 인스턴스는 아닙니다)는 불변입니다. 파이썬에서 형 객체의 어트리뷰트를 추가하거나 수정할 수 없습니다.
- 정적 형 객체는 서브 인터프리터에서 공유되므로, 서브 인터프리터 관련 상태를 포함하지 않아야 합니다.

Also, since *PyTypeObject* is only part of the *Limited API* as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

13.4.7 힙 형

An alternative to *static types* is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement. Heap types have the *Py_TPFLAGS_HEAPTYPE* flag set.

This is done by filling a *PyType_Spec* structure and calling *PyType_FromSpec()*, *PyType_FromSpecWithBases()*, *PyType_FromModuleAndSpec()*, or *PyType_FromMetaclass()*.

13.4.8 숫자 객체 구조체

type **PyNumberMethods**

이 구조체는 객체가 숫자 프로토콜을 구현하는 데 사용하는 함수에 대한 포인터를 담습니다. 각 함수는 숫자 프로토콜 섹션에서 설명하는 유사한 이름의 함수가 사용합니다.

구조체 정의는 다음과 같습니다:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
     binaryfunc nb_and;
     binaryfunc nb_xor;
     binaryfunc nb_or;
     unaryfunc nb_int;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    void *nb_reserved;
    unaryfunc nb_float;

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
    binaryfunc nb_inplace_rshift;
    binaryfunc nb_inplace_and;
    binaryfunc nb_inplace_xor;
    binaryfunc nb_inplace_or;

    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    unaryfunc nb_index;

    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

참고

이항과 삼항 함수는 모든 피연산자의 형을 확인하고, 필요한 변환을 구현해야합니다 (적어도 피연산자 중 하나는 정의된 형의 인스턴스입니다). 주어진 피연산자에 대해 연산이 정의되지 않으면, 이항과 삼항 함수는 `Py_NotImplemented`를 반환해야하며, 다른 에러가 발생하면 `NULL`을 반환하고 예외를 설정해야 합니다.

참고

The *nb_reserved* field should always be `NULL`. It was previously called `nb_long`, and was renamed in Python 3.0.1.

binaryfunc *PyNumberMethods*.**nb_add**

binaryfunc *PyNumberMethods*.**nb_subtract**

binaryfunc *PyNumberMethods*.**nb_multiply**

binaryfunc *PyNumberMethods*.**nb_remainder**

binaryfunc *PyNumberMethods*.**nb_divmod**

ternaryfunc *PyNumberMethods*.**nb_power**

unaryfunc *PyNumberMethods*.**nb_negative**

unaryfunc *PyNumberMethods*.**nb_positive**

unaryfunc *PyNumberMethods*.**nb_absolute**

inquiry *PyNumberMethods*.**nb_bool**

unaryfunc *PyNumberMethods*.**nb_invert**

binaryfunc *PyNumberMethods*.**nb_lshift**

binaryfunc *PyNumberMethods*.**nb_rshift**

binaryfunc *PyNumberMethods*.**nb_and**

binaryfunc *PyNumberMethods*.**nb_xor**

binaryfunc *PyNumberMethods*.**nb_or**

unaryfunc *PyNumberMethods*.**nb_int**

void **PyNumberMethods*.**nb_reserved**

unaryfunc *PyNumberMethods*.**nb_float**

binaryfunc *PyNumberMethods*.**nb_inplace_add**

binaryfunc *PyNumberMethods*.**nb_inplace_subtract**

binaryfunc *PyNumberMethods*.**nb_inplace_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_remainder**

ternaryfunc *PyNumberMethods*.**nb_inplace_power**

binaryfunc *PyNumberMethods*.**nb_inplace_lshift**

binaryfunc *PyNumberMethods*.**nb_inplace_rshift**

binaryfunc *PyNumberMethods*.**nb_inplace_and**

binaryfunc *PyNumberMethods*.**nb_inplace_xor**

binaryfunc *PyNumberMethods*.**nb_inplace_or**

binaryfunc *PyNumberMethods*.**nb_floor_divide**

binaryfunc *PyNumberMethods*.**nb_true_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_floor_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_true_divide**

unaryfunc *PyNumberMethods*.**nb_index**

binaryfunc *PyNumberMethods*.**nb_matrix_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_matrix_multiply**

13.4.9 매핑 객체 구조체

type **PyMappingMethods**

이 구조체에는 객체가 매핑 프로토콜을 구현하는 데 사용하는 함수에 대한 포인터를 담습니다. 세 개의 멤버가 있습니다:

lenfunc *PyMappingMethods*.**mp_length**

이 함수는 *PyMapping_Size()*와 *PyObject_Size()*에서 사용되며, 같은 서명을 갖습니다. 객체에 길이가 정의되어 있지 않으면 이 슬롯을 NULL로 설정할 수 있습니다.

binaryfunc *PyMappingMethods*.**mp_subscript**

이 함수는 *PyObject_GetItem()*과 *PySequence_GetSlice()*에서 사용되며, PyObject_GetItem() 과 같은 서명을 갖습니다. *PyMapping_Check()* 함수가 1을 반환하려면, 이 슬롯을 채워야합니다, 그렇지 않으면 NULL일 수 있습니다.

objobjargproc *PyMappingMethods*.**mp_ass_subscript**

This function is used by *PyObject_SetItem()*, *PyObject_DelItem()*, *PySequence_SetSlice()* and *PySequence_DelSlice()*. It has the same signature as PyObject_SetItem(), but *v* can also be set to NULL to delete an item. If this slot is NULL, the object does not support item assignment and deletion.

13.4.10 시퀀스 객체 구조체

type **PySequenceMethods**

이 구조체는 객체가 시퀀스 프로토콜을 구현하는 데 사용하는 함수에 대한 포인터를 담습니다.

lenfunc *PySequenceMethods*.**sq_length**

이 함수는 *PySequence_Size()*와 *PyObject_Size()*에서 사용되며, 같은 서명을 갖습니다. 또한 *sq_item*과 *sq_ass_item* 슬롯을 통해 음수 인덱스를 처리하는 데 사용됩니다.

binaryfunc *PySequenceMethods*.**sq_concat**

이 함수는 *PySequence_Concat()*에서 사용되며 같은 서명을 갖습니다. *nb_add* 슬롯을 통해 숫자 덧셈을 시도한 후, + 연산자에서도 사용됩니다.

ssizeargfunc *PySequenceMethods*.**sq_repeat**

이 함수는 *PySequence_Repeat()*에서 사용되며 같은 서명을 갖습니다. *nb_multiply* 슬롯을 통해 숫자 곱셈을 시도한 후, * 연산자에서도 사용됩니다.

ssizeargfunc *PySequenceMethods*.**sq_item**

이 함수는 *PySequence_GetItem()*에서 사용되며 같은 서명을 갖습니다. *mp_subscript* 슬롯을 통해 서브스크립션(subscription)을 시도한 후, *PyObject_GetItem()*에서도 사용됩니다. *PySequence_Check()* 함수가 1을 반환하려면, 이 슬롯을 채워야합니다, 그렇지 않으면 NULL일 수 있습니다.

Negative indexes are handled as follows: if the *sq_length* slot is filled, it is called and the sequence length is used to compute a positive index which is passed to *sq_item*. If sq_length is NULL, the index is passed as is to the function.

ssizeobjargproc *PySequenceMethods*.**sq_ass_item**

이 함수는 *PySequence_SetItem()*에서 사용되며 같은 서명을 갖습니다. *mp_ass_subscript* 슬롯을 통해 항목 대입과 삭제를 시도한 후, *PyObject_SetItem()*과 *PyObject_DelItem()*에서도 사용됩니다. 객체가 항목 대입과 삭제를 지원하지 않으면 이 슬롯은 NULL로 남겨 둘 수 있습니다.

objobjproc *PySequenceMethods*.**sq_contains**

이 함수는 *PySequence_Contains()*에서 사용될 수 있으며 같은 서명을 갖습니다. 이 슬롯은 NULL로 남겨 둘 수 있습니다, 이때 PySequence_Contains()는 일치하는 것을 찾을 때까지 시퀀스를 단순히 탐색합니다.

binaryfunc *PySequenceMethods*.**sq_inplace_concat**

이 함수는 *PySequence_InPlaceConcat()*에서 사용되며 같은 서명을 갖습니다. 첫 번째 피연산자를 수정하고 그것을 반환해야 합니다. 이 슬롯은 NULL로 남겨 둘 수 있으며, 이때 PySequence_InPlaceConcat()은 *PySequence_Concat()*으로 폴백 됩니다. *nb_inplace_add* 슬롯을 통해 숫자 제자리 덧셈을 시도한 후, 증분 대입 +=에서 사용됩니다.

ssizeargfunc *PySequenceMethods*.**sq_inplace_repeat**

이 함수는 *PySequence_InPlaceRepeat()*에서 사용되며 같은 서명을 갖습니다. 첫 번째 피연산자를 수정하고 그것을 반환해야 합니다. 이 슬롯은 NULL로 남겨 둘 수 있으며, 이때 PySequence_InPlaceRepeat()는 *PySequence_Repeat()*로 폴백 됩니다. *nb_inplace_multiply* 슬롯을 통해 숫자 제자리 곱셈을 시도한 후, 증분 대입 *=에서도 사용됩니다.

13.4.11 버퍼 객체 구조체

type **PyBufferProcs**

이 구조체는 버퍼 프로토콜에 필요한 함수에 대한 포인터를 담습니다. 프로토콜은 제공자(exporter) 객체가 내부 데이터를 소비자 객체에 노출하는 방법을 정의합니다.

getbufferproc *PyBufferProcs*.**bf_getbuffer**

이 함수의 서명은 다음과 같습니다:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

*view*를 채우기 위해 *exporter*에 대한 *flags*에 지정된 요청을 처리합니다. 포인트 (3) 을 제외하고, 이 함수의 구현은 다음 단계를 반드시 수행해야 합니다:

(1) Check if the request can be met. If not, raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

(2) 요청된 필드를 채웁니다.

(3) 내보내기 횟수에 대한 내부 카운터를 증가시킵니다.

(4) Set `view->obj` to *exporter* and increment `view->obj`.

(5) `0`을 반환합니다.

*exporter*가 버퍼 공급자의 체인이나 트리의 일부이면, 두 가지 주요 체계를 사용할 수 있습니다:

- Re-export: Each member of the tree acts as the exporting object and sets `view->obj` to a new reference to itself.
- Redirect: The buffer request is redirected to the root object of the tree. Here, `view->obj` will be a new reference to the root object.

*view*의 개별 필드는 섹션 버퍼 구조체에 설명되어 있으며, 제공자가 특정 요청에 응답해야 하는 규칙은 섹션 버퍼 요청 유형에 있습니다.

Py_buffer 구조체에서 가리키는 모든 메모리는 제공자에게 속하며 남은 소비자가 없어질 때까지 유효해야 합니다. *format*, *shape*, *strides*, *suboffsets* 및 *internal*은 소비자에게는 읽기 전용입니다.

*PyBuffer_FillInfo()*는 모든 요청 유형을 올바르게 처리하면서 간단한 바이트열 버퍼를 쉽게 노출할 수 있는 방법을 제공합니다.

*PyObject_GetBuffer()*는 이 함수를 감싸는 소비자 용 인터페이스입니다.

releasebufferproc *PyBufferProcs*.**bf_releasebuffer**

이 함수의 서명은 다음과 같습니다:

```
void (PyObject *exporter, Py_buffer *view);
```

버퍼 자원 해제 요청을 처리합니다. 자원을 해제할 필요가 없으면, *PyBufferProcs.bf_releasebuffer*는 `NULL`일 수 있습니다. 그렇지 않으면, 이 함수의 표준 구현은 다음과 같은 선택적 단계를 수행합니다:

(1) 내보내기 횟수에 대한 내부 카운터를 줄입니다.

(2) 카운터가 `0`이면, *view*와 관련된 모든 메모리를 해제합니다.

제공자는 반드시 *internal* 필드를 사용하여 버퍼 특정 자원을 추적해야 합니다. 이 필드는 변경되지 않고 유지됨이 보장되지만, 소비자는 원래 버퍼의 사본을 *view* 인자로 전달할 수 있습니다.

This function MUST NOT decrement `view->obj`, since that is done automatically in *PyBuffer_Release()* (this scheme is useful for breaking reference cycles).

*PyBuffer_Release()*는 이 기능을 감싸는 소비자 용 인터페이스입니다.

13.4.12 비동기 객체 구조체

Added in version 3.5.

type **PyAsyncMethods**

이 구조체는 어웨이터블와 비동기 이터레이터 객체를 구현하는 데 필요한 함수에 대한 포인터를 담습니다.

구조체 정의는 다음과 같습니다:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc *PyAsyncMethods*.**am_await**

이 함수의 서명은 다음과 같습니다:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an *iterator*, i.e. *PyIter_Check()* must return `1` for it.

객체가 어웨이터블이 아니면 이 슬롯을 `NULL`로 설정할 수 있습니다.

unaryfunc *PyAsyncMethods*.**am_aiter**

이 함수의 서명은 다음과 같습니다:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

객체가 비동기 이터레이션 프로토콜을 구현하지 않으면 이 슬롯은 `NULL`로 설정될 수 있습니다.

unaryfunc *PyAsyncMethods*.**am_anext**

이 함수의 서명은 다음과 같습니다:

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to `NULL`.

sendfunc *PyAsyncMethods*.**am_send**

이 함수의 서명은 다음과 같습니다:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

See *PyIter_Send()* for details. This slot may be set to `NULL`.

Added in version 3.10.

13.4.13 슬롯 형 typedef

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

Part of the 안정 ABI. The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with *ob_refcnt* set to `1` and *ob_type* set to the type argument. If the type's *tp_itemsize* is non-zero, the object's *ob_size* field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, *nitems* is not used and the length of the block should be *tp_basicsize*.

이 함수는 다른 인스턴스 초기화를 수행하지 않아야 합니다, 추가 메모리를 할당도 안 됩니다; 그것은 *tp_new*에 의해 수행되어야 합니다.

typedef void (***destructor**)(*PyObject**)

Part of the 안정 ABI.

typedef void (***freefunc**)(void*)

*tp_free*를 참조하십시오.

typedef *PyObject* *(***newfunc**)(*PyTypeObject**, *PyObject**, *PyObject**)

Part of the 안정 ABI. *tp_new*를 참조하십시오.

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the 안정 ABI. *tp_init*를 참조하십시오.

typedef *PyObject* *(***reprfunc**)(*PyObject**)

Part of the 안정 ABI. *tp_repr*을 참조하십시오.

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

Part of the 안정 ABI. 객체의 명명된 어트리뷰트 값을 반환합니다.

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

Part of the 안정 ABI. 객체의 명명된 어트리뷰트 값을 설정합니다. 어트리뷰트를 삭제하려면 value 인자가 NULL로 설정됩니다.

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

Part of the 안정 ABI. 객체의 명명된 어트리뷰트 값을 반환합니다.

*tp_getattro*를 참조하십시오.

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Part of the 안정 ABI. 객체의 명명된 어트리뷰트 값을 설정합니다. 어트리뷰트를 삭제하려면 value 인자가 NULL로 설정됩니다.

*tp_setattro*를 참조하십시오.

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the 안정 ABI. See *tp_descr_get*.

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the 안정 ABI. See *tp_descr_set*.

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

Part of the 안정 ABI. *tp_hash*를 참조하십시오.

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

Part of the 안정 ABI. *tp_richcompare*를 참조하십시오.

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

Part of the 안정 ABI. *tp_iter*를 참조하십시오.

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

Part of the 안정 ABI. *tp_iternext*를 참조하십시오.

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

Part of the 안정 ABI.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

Part of the 안정 ABI 버전 *3.12* 이후로.

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

Part of the 안정 ABI 버전 *3.12* 이후로.

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

Part of the 안정 ABI.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)
Part of the 안정 ABI.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)
See *am_send*.

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)
Part of the 안정 ABI.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)
Part of the 안정 ABI.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)
Part of the 안정 ABI.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)
Part of the 안정 ABI.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)
Part of the 안정 ABI.

13.4.14 예

다음은 파이썬 형 정의의 간단한 예입니다. 여기에는 여러분이 만날 수 있는 일반적인 사용법이 포함됩니다. 일부는 까다로운 코너 사례를 보여줍니다. 더 많은 예제, 실용 정보 및 자습서는 defining-new-types와 new-types-topics를 참조하십시오.

A basic *static type*:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

더 상세한 초기화자를 사용하는 이전 코드(특히 CPython 코드 베이스에서)를 찾을 수도 있습니다:

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    (reprfunc)myobj_repr,           /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    0,                              /* tp_hash */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    0,                              /* tp_flags */
    PyDoc_STR("My objects"),        /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    0,                              /* tp_methods */
    0,                              /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    0,                              /* tp_init */
    0,                              /* tp_alloc */
    myobj_new,                      /* tp_new */
};
```

약한 참조, 인스턴스 딕셔너리 및 해싱을 지원하는 형:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
         Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
         Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag:

```
typedef struct {
    PyUnicodeObject raw;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest *static type* with fixed-length instances:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest *static type* with variable-length instances:

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

13.5 순환 가비지 수집 지원

순환 참조를 포함하는 가비지를 탐지하고 수집하는 파이썬의 지원은 역시 컨테이너일 수 있는 다른 객체의 "컨테이너" 인 객체 형의 지원이 필요합니다. 다른 객체에 대한 참조를 저장하지 않거나, 원자 형(가령 숫자나 문자열)에 대한 참조만 저장하는 형은 가비지 수집에 대한 어떤 명시적인 지원을 제공할 필요가 없습니다.

컨테이너형을 만들려면, 형 객체의 *tp_flags* 필드가 *Py_TPFLAGS_HAVE_GC*를 포함해야 하고 *tp_traverse* 처리기 구현을 제공해야 합니다. 형의 인스턴스가 가변이면, *tp_clear* 구현도 제공해야 합니다.

Py_TPFLAGS_HAVE_GC
: 이 플래그가 설정된 형의 객체는 여기에 설명된 규칙을 준수해야 합니다. 편의를 위해 이러한 객체를 컨테이너 객체라고 하겠습니다.

컨테이너형의 생성자는 두 가지 규칙을 준수해야 합니다:

1. 객체의 메모리는 *PyObject_GC_New* 나 *PyObject_GC_NewVar*를 사용하여 할당해야 합니다.
2. 다른 컨테이너에 대한 참조를 포함할 수 있는 모든 필드가 초기화되면, *PyObject_GC_Track()*를 호출해야 합니다.

마찬가지로, 객체의 할당해제자(deallocator)는 비슷한 규칙 쌍을 준수해야 합니다:

1. 다른 컨테이너를 참조하는 필드가 무효화 되기 전에, *PyObject_GC_UnTrack()*를 호출해야 합니다.
2. 객체의 메모리는 *PyObject_GC_Del()*를 사용하여 할당 해제되어야 합니다.

> **경고**
>
> If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a *tp_traverse* handler or explicitly use one from its subclass or subclasses.
>
> When calling *PyType_Ready()* or some of the APIs that indirectly call it like *PyType_FromSpecWithBases()* or *PyType_FromSpec()* the interpreter will automatically populate the *tp_flags*, *tp_traverse* and *tp_clear* fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the *Py_TPFLAGS_HAVE_GC* flag.

PyObject_GC_New(TYPE, typeobj)

*PyObject_New*와 유사하지만, *Py_TPFLAGS_HAVE_GC* 플래그가 설정된 컨테이너 객체를 위한 것.

Do not call this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with *PyObject_GC_Del()* (usually called via the object's *tp_free* slot).

> **더 보기**
>
> - *PyObject_GC_Del()*
> - *PyObject_New*
> - *PyType_GenericAlloc()*
> - *tp_alloc*

PyObject_GC_NewVar(TYPE, typeobj, size)

*PyObject_NewVar*와 유사하지만, *Py_TPFLAGS_HAVE_GC* 플래그가 설정된 컨테이너 객체를 위한 것.

Do not call this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with *PyObject_GC_Del()* (usually called via the object's *tp_free* slot).

> **더 보기**
>
> - *PyObject_GC_Del()*
> - *PyObject_NewVar*
> - *PyType_GenericAlloc()*
> - *tp_alloc*

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

이것은 불안정 *API*. It may change without warning in minor releases.

Analogous to `PyObject_GC_New` but allocates *extra_size* bytes at the end of the object (at offset `tp_basicsize`). The allocated memory is initialized to zeros, except for the `Python object header`.

The extra data will be deallocated with the object, but otherwise it is not managed by Python.

Memory allocated by this function must be freed with `PyObject_GC_Del()` (usually called via the object's `tp_free` slot).

경고

The function is marked as unstable because the final mechanism for reserving extra data after an instance is not yet decided. For allocating a variable number of fields, prefer using `PyVarObject` and `tp_itemsize` instead.

Added in version 3.12.

PyObject_GC_Resize(TYPE, op, newsize)

`PyObject_NewVar`에 의해 할당된 객체의 크기를 변경합니다. 형 `TYPE*` (모든 C 형을 나타냅니다) 의 크기가 조정된 객체나 실패하면 `NULL`을 반환합니다.

*op*는 형 `PyVarObject*`이어야 하고 아직 수거기가 추적하지 않아야 합니다. *newsize*는 형 `Py_ssize_t`이어야 합니다.

void **PyObject_GC_Track**(*PyObject* *op)

Part of the 안정 ABI. 수거기가 추적하는 컨테이너 객체 집합에 객체 *op*를 추가합니다. 수거기는 예기치 않은 시간에 실행될 수 있으므로 추적되는 동안 객체가 유효해야 합니다. `tp_traverse` 처리기가 탐색하는 모든 필드가 유효해지면 호출해야 합니다, 보통 생성자의 끝부분 근처입니다.

int **PyObject_IS_GC**(*PyObject* *obj)

객체가 가비지 수거기 프로토콜을 구현하면 0이 아닌 값을 반환하고, 그렇지 않으면 0을 반환합니다.

이 함수가 0을 반환하면 가비지 수거기가 객체를 추적할 수 없습니다.

int **PyObject_GC_IsTracked**(*PyObject* *op)

Part of the 안정 ABI 버전 *3.9* 이후로. *op*의 객체 형이 GC 프로토콜을 구현하고 *op*가 현재 가비지 수거기가 추적 중이면 1을 반환하고 그렇지 않으면 0을 반환합니다.

이것은 파이썬 함수 `gc.is_tracked()`에 해당합니다.

Added in version 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

Part of the 안정 ABI 버전 *3.9* 이후로. *op*의 객체 형이 GC 프로토콜을 구현하고 가비지 수거기가 *op*를 이미 파이널라이즈 했으면 1을 반환하고 그렇지 않으면 0을 반환합니다.

이것은 파이썬 함수 `gc.is_finalized()`에 해당합니다.

Added in version 3.9.

void **PyObject_GC_Del**(void *op)

Part of the 안정 ABI. `PyObject_GC_New`나 `PyObject_GC_NewVar`를 사용하여 객체에 할당된 메모리를 해제합니다.

Do not call this directly to free an object's memory; call the type's `tp_free` slot instead.

Do not use this for memory allocated by `PyObject_New`, `PyObject_NewVar`, or related allocation functions; use `PyObject_Free()` instead.

더 보기

- *PyObject_Free()* is the non-GC equivalent of this function.
- *PyObject_GC_New*
- *PyObject_GC_NewVar*
- *PyType_GenericAlloc()*
- *tp_free*

void **PyObject_GC_UnTrack**(void *op)

Part of the 안정 ABI. 수거기가 추적하는 컨테이너 객체 집합에서 *op* 객체를 제거합니다. *PyObject_GC_Track()*를 이 객체에 대해 다시 호출하여 추적 객체 집합에 다시 추가할 수 있음에 유의하십시오. 할당해제자(*tp_dealloc* 처리기)는 *tp_traverse* 처리기에서 사용하는 필드가 무효화되기 전에 객체에 대해 이 함수를 호출해야 합니다.

버전 3.8에서 변경: `_PyObject_GC_TRACK()`과 `_PyObject_GC_UNTRACK()` 매크로는 공용 C API에서 제거되었습니다.

tp_traverse 처리기는 다음과 같은 형의 함수 매개 변수를 받아들입니다:

typedef int (***visitproc**)(*PyObject* *object, void *arg)

Part of the 안정 ABI. *tp_traverse* 처리기에 전달되는 방문자 함수의 형. 이 함수는 탐색하는 객체를 *object*로, *tp_traverse* 처리기의 세 번째 매개 변수를 *arg*로 호출되어야 합니다. 파이썬 코어는 순환 가비지 탐지를 구현하기 위해 여러 방문자 함수를 사용합니다; 사용자가 자신의 방문자 함수를 작성해야 할 필요는 없습니다.

tp_traverse 처리기는 다음 형이어야 합니다:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

Part of the 안정 ABI. 컨테이너 객체의 탐색 함수입니다. 구현은 *self*에 직접 포함된 각 객체에 대해 *visit* 함수를 호출해야 하며, *visit*에 대한 매개 변수는 포함된 객체와 처리기로 전달된 *arg* 값입니다. *visit* 함수는 `NULL` object 인자로 호출하면 안 됩니다. *visit*가 0이 아닌 값을 반환하면 그 값이 즉시 반환되어야 합니다.

tp_traverse 처리기 작성을 단순화하기 위해, *Py_VISIT()* 매크로가 제공됩니다. 이 매크로를 사용하려면, *tp_traverse* 구현은 인자의 이름을 정확히 *visit* 와 *arg*로 지정해야 합니다:

Py_VISIT(o)

If the *PyObject** *o* is not `NULL`, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, *tp_traverse* handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

tp_clear 처리기는 *inquiry* 형이거나 객체가 불변이면 `NULL`이어야 합니다.

typedef int (***inquiry**)(*PyObject* *self)

Part of the 안정 ABI. 참조 순환을 생성했을 수 있는 참조를 삭제합니다. 불변 객체는 참조 순환을 직접 생성할 수 없으므로, 이 메서드를 정의 할 필요가 없습니다. 이 메서드를 호출한 후에도 객체가 유효해야 합니다 (단지 참조에 대해 *Py_DECREF()*를 호출하지 마십시오). 이 객체가 참조 순환에 참여하고 있음을 수거기가 감지하면 이 메서드를 호출합니다.

13.5.1 가비지 수거기 상태 제어하기

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t **PyGC_Collect**(void)

Part of the 안정 ABI. Perform a full garbage collection, if the garbage collector is enabled. (Note that `gc.collect()` runs it unconditionally.)

Returns the number of collected + unreachable objects which cannot be collected. If the garbage collector is disabled or already collecting, returns `0` immediately. Errors during garbage collection are passed to `sys.unraisablehook`. This function does not raise exceptions.

int **PyGC_Enable**(void)

Part of the 안정 ABI 버전 *3.10* 이후로. Enable the garbage collector: similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int **PyGC_Disable**(void)

Part of the 안정 ABI 버전 *3.10* 이후로. Disable the garbage collector: similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int **PyGC_IsEnabled**(void)

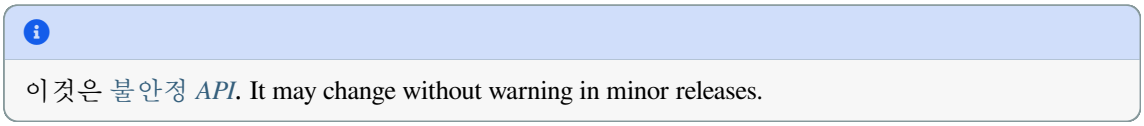
Part of the 안정 ABI 버전 *3.10* 이후로. Query the state of the garbage collector: similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

Added in version 3.10.

13.5.2 가비지 수거기 상태 조회하기

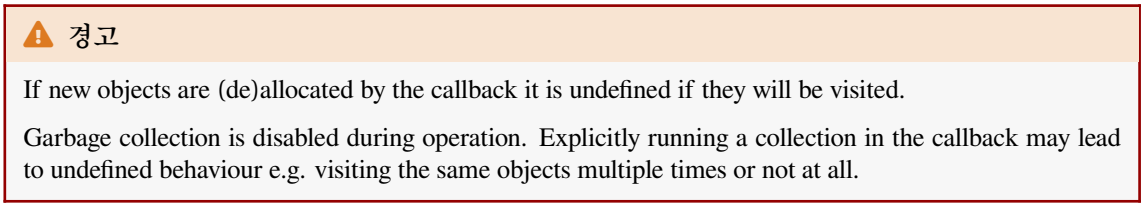
The C-API provides the following interface for querying information about the garbage collector.

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)



이것은 불안정 *API*. It may change without warning in minor releases.

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.



경고

If new objects are (de)allocated by the callback it is undefined if they will be visited.

Garbage collection is disabled during operation. Explicitly running a collection in the callback may lead to undefined behaviour e.g. visiting the same objects multiple times or not at all.

Added in version 3.12.

typedef int (***gcvisitobjects_t**)(*PyObject* *object, void *arg)

Type of the visitor function to be passed to *`PyUnstable_GC_VisitObjects()`*. *arg* is the same as the *arg* passed to `PyUnstable_GC_VisitObjects`. Return `1` to continue iteration, return `0` to stop iteration. Other return values are reserved for now so behavior on returning anything else is undefined.

Added in version 3.12.

CHAPTER 14

API와 ABI 버전 붙이기

14.1 Build-time version constants

CPython exposes its version number in the following macros. Note that these correspond to the version code is **built** with. See *Py_Version* for the version used at **run time**.

See *C API Stability* for a discussion of API and ABI stability across versions.

PY_MAJOR_VERSION

`3.4.1a2`의 `3`.

PY_MINOR_VERSION

`3.4.1a2`의 `4`.

PY_MICRO_VERSION

`3.4.1a2`의 `1`.

PY_RELEASE_LEVEL

`3.4.1a2`의 `a`. 알파는 `0xA`, 베타는 `0xB`, 배포 후보는 `0xC`, 최종은 `0xF` 가 될 수 있습니다.

PY_RELEASE_SERIAL

`3.4.1a2`의 `2`. 최종 배포는 0.

PY_VERSION_HEX

The Python version number encoded in a single integer. See *Py_PACK_FULL_VERSION()* for the encoding details.

Use this for numeric comparisons, for example, `#if PY_VERSION_HEX >= ...`.

14.2 Run-time version

const unsigned long **Py_Version**

Part of the 안정 ABI *버전 3.11 이후로.* The Python runtime version number encoded in a single constant integer. See *Py_PACK_FULL_VERSION()* for the encoding details. This contains the Python version used at run time.

Use this for numeric comparisons, for example, `if (Py_Version >= ...)`.

Added in version 3.11.

14.3 Bit-packing macros

uint32_t **Py_PACK_FULL_VERSION**(int major, int minor, int micro, int release_level, int release_serial)

Part of the 안정 ABI *버전 3.14 이후로.* Return the given version, encoded as a single 32-bit integer with the following structure:

Argument	No. of bits	Bit mask	Bit shift	Example values	
				`3.4.1a2`	`3.10.0`
major	8	`0xFF000000`	24	`0x03`	`0x03`
minor	8	`0x00FF0000`	16	`0x04`	`0x0A`
micro	8	`0x0000FF00`	8	`0x01`	`0x00`
release_level	4	`0x000000F0`	4	`0xA`	`0xF`
release_serial	4	`0x0000000F`	0	`0x2`	`0x0`

For example:

Version	`Py_PACK_FULL_VERSION` arguments	Encoded version
`3.4.1a2`	`(3, 4, 1, 0xA, 2)`	`0x030401a2`
`3.10.0`	`(3, 10, 0, 0xF, 0)`	`0x030a00f0`

Out-of range bits in the arguments are ignored. That is, the macro can be defined as:

```
#ifndef Py_PACK_FULL_VERSION
#define Py_PACK_FULL_VERSION(X, Y, Z, LEVEL, SERIAL) ( \
   (((X) & 0xff) << 24) |                              \
   (((Y) & 0xff) << 16) |                              \
   (((Z) & 0xff) << 8) |                               \
   (((LEVEL) & 0xf) << 4) |                            \
   (((SERIAL) & 0xf) << 0))
#endif
```

`Py_PACK_FULL_VERSION` is primarily a macro, intended for use in `#if` directives, but it is also available as an exported function.

Added in version 3.14.

uint32_t **Py_PACK_VERSION**(int major, int minor)

Part of the 안정 ABI *버전 3.14 이후로.* Equivalent to `Py_PACK_FULL_VERSION(major, minor, 0, 0, 0)`. The result does not correspond to any Python release, but is useful in numeric comparisons.

Added in version 3.14.

CHAPTER 15

Monitoring C API

Added in version 3.13.

An extension may need to interact with the event monitoring system. Subscribing to events and registering callbacks can be done via the Python API exposed in `sys.monitoring`.

CHAPTER 16

Generating Execution Events

The functions below make it possible for an extension to fire monitoring events as it emulates the execution of Python code. Each of these functions accepts a `PyMonitoringState` struct which contains concise information about the activation state of events, as well as the event arguments, which include a `PyObject*` representing the code object, the instruction offset and sometimes additional, event-specific arguments (see `sys.monitoring` for details about the signatures of the different event callbacks). The `codelike` argument should be an instance of `types.CodeType` or of a type that emulates it.

The VM disables tracing when firing an event, so there is no need for user code to do that.

Monitoring functions should not be called with an exception set, except those listed below as working with the current exception.

type **PyMonitoringState**

Representation of the state of an event type. It is allocated by the user while its contents are maintained by the monitoring API functions described below.

All of the functions below return 0 on success and -1 (with an exception set) on error.

See `sys.monitoring` for descriptions of the events.

int **PyMonitoring_FirePyStartEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_START` event.

int **PyMonitoring_FirePyResumeEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_RESUME` event.

int **PyMonitoring_FirePyReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_RETURN` event.

int **PyMonitoring_FirePyYieldEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_YIELD` event.

int **PyMonitoring_FireCallEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *callable, *PyObject* *arg0)

Fire a `CALL` event.

int **PyMonitoring_FireLineEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, int lineno)

Fire a `LINE` event.

int **PyMonitoring_FireJumpEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `JUMP` event.

int **PyMonitoring_FireBranchLeftEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `BRANCH_LEFT` event.

int **PyMonitoring_FireBranchRightEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `BRANCH_RIGHT` event.

int **PyMonitoring_FireCReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `C_RETURN` event.

int **PyMonitoring_FirePyThrowEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_THROW` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireCRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `C_RAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireReraiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RERAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireExceptionHandledEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire an `EXCEPTION_HANDLED` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FirePyUnwindEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_UNWIND` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireStopIterationEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *value)

Fire a `STOP_ITERATION` event. If `value` is an instance of `StopIteration`, it is used. Otherwise, a new `StopIteration` instance is created with `value` as its argument.

16.1 Managing the Monitoring State

Monitoring states can be managed with the help of monitoring scopes. A scope would typically correspond to a python function.

int **PyMonitoring_EnterScope**(*PyMonitoringState* *state_array, uint64_t *version, const uint8_t *event_types, *Py_ssize_t* length)

Enter a monitored scope. `event_types` is an array of the event IDs for events that may be fired from the scope. For example, the ID of a `PY_START` event is the value `PY_MONITORING_EVENT_PY_START`, which is numerically equal to the base-2 logarithm of `sys.monitoring.events.PY_START`. `state_array` is an array with a monitoring state entry for each event in `event_types`, it is allocated by the user but populated by `PyMonitoring_EnterScope()` with information about the activation state of the event. The size of `event_types` (and hence also of `state_array`) is given in `length`.

The `version` argument is a pointer to a value which should be allocated by the user together with `state_array` and initialized to 0, and then set only by `PyMonitoring_EnterScope()` itself. It allows this function to determine whether event states have changed since the previous call, and to return quickly if they have not.

The scopes referred to here are lexical scopes: a function, class or method. `PyMonitoring_EnterScope()` should be called whenever the lexical scope is entered. Scopes can be reentered, reusing the same *state_array* and *version*, in situations like when emulating a recursive Python function. When a code-like' s execution is paused, such as when emulating a generator, the scope needs to be exited and re-entered.

The macros for *event_types* are:

Macro	Event
PY_MONITORING_EVENT_BRANCH_LEFT	BRANCH_LEFT
PY_MONITORING_EVENT_BRANCH_RIGHT	BRANCH_RIGHT
PY_MONITORING_EVENT_CALL	CALL
PY_MONITORING_EVENT_C_RAISE	C_RAISE
PY_MONITORING_EVENT_C_RETURN	C_RETURN
PY_MONITORING_EVENT_EXCEPTION_HANDLED	EXCEPTION_HANDLED
PY_MONITORING_EVENT_INSTRUCTION	INSTRUCTION
PY_MONITORING_EVENT_JUMP	JUMP
PY_MONITORING_EVENT_LINE	LINE
PY_MONITORING_EVENT_PY_RESUME	PY_RESUME
PY_MONITORING_EVENT_PY_RETURN	PY_RETURN
PY_MONITORING_EVENT_PY_START	PY_START
PY_MONITORING_EVENT_PY_THROW	PY_THROW
PY_MONITORING_EVENT_PY_UNWIND	PY_UNWIND
PY_MONITORING_EVENT_PY_YIELD	PY_YIELD
PY_MONITORING_EVENT_RAISE	RAISE
PY_MONITORING_EVENT_RERAISE	RERAISE
PY_MONITORING_EVENT_STOP_ITERATION	STOP_ITERATION

int **PyMonitoring_ExitScope**(void)

Exit the last scope that was entered with `PyMonitoring_EnterScope()`.

int **PY_MONITORING_IS_INSTRUMENTED_EVENT**(uint8_t ev)

Return true if the event corresponding to the event ID *ev* is a local event.

Added in version 3.13.

버전 3.14부터 폐지됨: This function is *soft deprecated*.

APPENDIX A

용어집

>>>
: 대화형 셸의 기본 파이썬 프롬프트. 인터프리터에서 대화형으로 실행될 수 있는 코드 예에서 자주 볼 수 있습니다.

...
: 다음과 같은 것들을 가리킬 수 있습니다:

  - 들여쓰기 된 코드 블록의 코드를 입력할 때, 쌍을 이루는 구분자 (괄호, 대괄호, 중괄호) 안에 코드를 입력할 때, 데코레이터 지정 후의 대화형 셸의 기본 파이썬 프롬프트.
  - The three dots form of the Ellipsis object.

abstract base class (추상 베이스 클래스)
: 추상 베이스 클래스는 `hasattr()` 같은 다른 테크닉들이 불편하거나 미묘하게 잘못된 (예를 들어, 매직 메서드) 경우, 인터페이스를 정의하는 방법을 제공함으로써 덕 타이핑 을 보완합니다. ABC는 가상 서브 클래스를 도입하는데, 클래스를 계승하지 않으면서도 `isinstance()` 와 `issubclass()` 에 의해 감지될 수 있는 클래스들입니다; `abc` 모듈 설명서를 보세요. 파이썬에는 많은 내장 ABC 들이 따라오는데 다음과 같은 것들이 있습니다: 자료 구조 (`collections.abc` 모듈에서), 숫자 (`numbers` 모듈에서), 스트림 (`io` 모듈에서), 임포트 파인더와 로더 (`importlib.abc` 모듈에서). `abc` 모듈을 사용해서 자신만의 ABC를 만들 수도 있습니다.

annotate function
: A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

annotation (어노테이션)
: 관습에 따라 형 힌트 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 반환 값과 연결된 레이블입니다.

  Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

  See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument (인자)
: 함수를 호출할 때 함수 (또는 메서드) 로 전달되는 값. 두 종류의 인자가 있습니다:

- 키워드 인자 (*keyword argument*): 함수 호출 때 식별자가 앞에 붙은 인자 (예를 들어, `name=`) 또는 `**` 를 앞에 붙인 딕셔너리로 전달되는 인자. 예를 들어, 다음과 같은 `complex()` 호출에서 `3` 과 `5` 는 모두 키워드 인자입니다:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 위치 인자 (*positional argument*): 키워드 인자가 아닌 인자. 위치 인자들은 인자 목록의 처음에 나오거나 이터러블 의 앞에 `*` 를 붙여 전달할 수 있습니다. 예를 들어, 다음과 같은 호출에서 `3` 과 `5` 는 모두 위치 인자입니다.

```
complex(3, 5)
complex(*(3, 5))
```

인자는 함수 바디의 이름 붙은 지역 변수에 대입됩니다. 이 대입에 적용되는 규칙들에 대해서는 calls 절을 보세요. 문법적으로, 어떤 표현식이건 인자로 사용될 수 있습니다; 구해진 값이 지역 변수에 대입됩니다.

용어집의 매개변수 항목과 FAQ 질문 인자와 매개변수의 차이 와 **PEP 362**도 보세요.

asynchronous context manager (비동기 컨텍스트 관리자)
`__aenter__()` 와 `__aexit__()` 메서드를 정의함으로써 `async with` 문에서 보이는 환경을 제어하는 객체. **PEP 492**로 도입되었습니다.

asynchronous generator (비동기 제너레이터)
비동기 제너레이터 이터레이터 를 돌려주는 함수. `async def` 로 정의되는 코루틴 함수처럼 보이는데, `async for` 루프가 사용할 수 있는 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다.

보통 비동기 제너레이터 함수를 가리키지만, 어떤 문맥에서는 비동기 제너레이터 이터레이터 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

비동기 제너레이터 함수는 `await` 표현식과, `async for` 문과, `async with` 문을 포함할 수 있습니다.

asynchronous generator iterator (비동기 제너레이터 이터레이터)
An object created by an *asynchronous generator* function.

비동기 이터레이터 인데 `__anext__()` 를 호출하면 어웨이터블 객체를 돌려주고, 이것은 다음 `yield` 표현식 까지 비동기 제너레이터 함수의 바디를 실행합니다.

각 `yield`는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 이터레이터 가 `__anext__()` 가 돌려주는 또 하나의 어웨이터블로 재개되면, 떠난 곳으로 복귀합니다. **PEP 492**와 **PEP 525**를 보세요.

asynchronous iterable (비동기 이터러블)
`async for` 문에서 사용될 수 있는 객체. `__aiter__()` 메서드는 비동기 이터레이터 를 돌려줘야 합니다. **PEP 492** 로 도입되었습니다.

asynchronous iterator (비동기 이터레이터)
`__aiter__()` 와 `__anext__()` 메서드를 구현하는 객체. `__anext__()` 는 어웨이터블 객체를 돌려줘야 합니다. `async for`는 `StopAsyncIteration` 예외가 발생할 때까지 비동기 이터레이터의 `__anext__()` 메서드가 돌려주는 어웨이터블을 풉니다. **PEP 492**로 도입되었습니다.

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds

of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

attribute (어트리뷰트)
흔히 점표현식을 사용하는 이름으로 참조되는 객체와 결합한 값. 예를 들어, 객체 *o*가 어트리뷰트 *a*를 가지면, *o.a*처럼 참조됩니다.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

awaitable (어웨이터블)
`await` 표현식에 사용할 수 있는 객체. 코루틴 이나 `__await__()` 메서드를 가진 객체가 될 수 있습니다. **PEP 492**를 보세요.

BDFL
자비로운 종신 독재자 (Benevolent Dictator For Life), 즉 Guido van Rossum, 파이썬의 창시자.

binary file (바이너리 파일)
바이트열류 객체들을 읽고 쓸 수 있는 파일 객체. 바이너리 파일의 예로는 바이너리 모드 (`'rb'`, `'wb'` 또는 `'rb+'`) 로 열린 파일, `sys.stdin.buffer`, `sys.stdout.buffer`, `io.BytesIO` 와 `gzip.GzipFile` 의 인스턴스를 들 수 있습니다.

`str` 객체를 읽고 쓸 수 있는 파일 객체에 대해서는 텍스트 파일 도 참조하세요.

borrowed reference (빌린 참조)
In Python' s C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling *`Py_INCREF()`* on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The *`Py_NewRef()`* function can be used to create a new *strong reference*.

bytes-like object (바이트열류 객체)
버퍼 프로토콜 를 지원하고 C-연속 버퍼를 익스포트 할 수 있습니다. 여러 공통 `memoryview` 객체들은 물론이고 `bytes`, `bytearray`, `array.array` 객체들을 포함합니다. 바이트열류 객체들은 바이너리 데이터를 다루는 여러 가지 연산들에 사용될 수 있습니다; 압축, 바이너리 파일로 저장, 소켓을 통한 전송 같은 것들이 있습니다.

어떤 연산들은 바이너리 데이터가 가변적일 필요가 있습니다. 이런 경우에 설명서는 종종 "읽고-쓰기 바이트열류 객체"라고 표현합니다. 가변 버퍼 객체의 예로는 `bytearray` 와 `bytearray` 의 `memoryview` 가 있습니다. 다른 연산들은 바이너리 데이터가 불변 객체 ("읽기 전용 바이트열류 객체")에 저장되도록 요구합니다; 이런 것들의 예로는 `bytes`와 `bytes` 객체의 `memoryview` 가 있습니다.

bytecode (바이트 코드)
파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현입니다. 바이트 코드는 `.pyc` 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만듭니다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있습니다). 이 "중간 언어" 는 각 바이트 코드에 대응하는 기계를 실행하는 가상 기계 에서 실행된다고 말합니다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 합니다.

바이트 코드 명령어들의 목록은 dis 모듈 설명서에 나옵니다.

callable (콜러블)
A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback (콜백)
인자로 전달되는 미래의 어느 시점에서 실행될 서브 루틴 함수.

class (클래스)
사용자 정의 객체들을 만들기 위한 주형. 클래스 정의는 보통 클래스의 인스턴스를 대상으로 연산하는 메서드 정의들을 포함합니다.

class variable (클래스 변수)
클래스에서 정의되고 클래스 수준 (즉, 클래스의 인스턴스에서가 아니라) 에서만 수정되는 변수.

closure variable (클로저 변수)
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number (복소수)
익숙한 실수 시스템의 확장인데, 모든 숫자가 실수부와 허수부의 합으로 표현됩니다. 허수부는 실수에 허수 단위(`-1`의 제곱근)를 곱한 것인데, 종종 수학에서는 `i`로, 공학에서는 `j`로 표기합니다. 파이썬은 후자의 표기법을 쓰는 복소수를 기본 지원합니다; 허수부는 `j` 접미사를 붙여서 표기합니다, 예를 들어, `3+1j`. `math` 모듈의 복소수 버전이 필요하면, `cmath`를 사용합니다. 복소수의 활용은 꽤 수준 높은 수학적 기능입니다. 필요하다고 느끼지 못한다면, 거의 확실히 무시해도 좋습니다.

context (컨텍스트)
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol (컨텍스트 관리 프로토콜)
The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

context manager (컨텍스트 관리자)
컨텍스트 관리 프로토콜을 구현하고 `with` 문에서 보이는 환경을 제어하는 객체. **PEP 343**을 참조하십시오.

context variable (컨텍스트 변수)
A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous (연속)
버퍼는 정확히 *C*-연속(*C-contiguous*)이거나 포트란 연속(*Fortran contiguous*)일 때 연속이라고 여겨집니다. 영차원 버퍼는 C-연속이면서 포트란 연속입니다. 일차원 배열에서, 항목들은 서로에 인접하고, 0에서 시작하는 오름차순 인덱스의 순서대로 메모리에 배치되어야 합니다. 다차원 C-연속 배열에서, 메모리 주소의 순서대로 항목들을 방문할 때 마지막 인덱스가 가장 빨리 변합니다. 하지만, 포트란 연속 배열에서는, 첫 번째 인덱스가 가장 빨리 변합니다.

coroutine (코루틴)
코루틴은 서브루틴의 더 일반화된 형태입니다. 서브루틴은 한 지점에서 진입하고 다른 지점에서 탈출합니다. 코루틴은 여러 다른 지점에서 진입하고, 탈출하고, 재개할 수 있습니다. 이것들은 `async def` 문으로 구현할 수 있습니다. **PEP 492**를 보세요.

coroutine function (코루틴 함수)
코루틴 객체를 돌려주는 함수. 코루틴 함수는 `async def` 문으로 정의될 수 있고, `await` 와 `async for`와 `async with` 키워드를 포함할 수 있습니다. 이것들은 **PEP 492** 에 의해 도입되었습니다.

CPython
파이썬 프로그래밍 언어의 규범적인 구현인데, python.org에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 "CPython" 이 사용됩니다.

current context (현재 컨텍스트)
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

cyclic isolate
A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the *cyclic garbage collector* is to identify these groups and break the reference cycles so that the memory can be reclaimed.

decorator (데코레이터)
다른 함수를 돌려주는 함수인데, 보통 `@wrapper` 문법을 사용한 함수 변환으로 적용됩니다. 데코레이터의 흔한 예는 `classmethod()` 과 `staticmethod()` 입니다.

데코레이터 문법은 단지 편의 문법일 뿐입니다. 다음 두 함수 정의는 의미상으로 동등합니다:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의 와 클래스 정의 의 설명서를 보면 됩니다.

descriptor (디스크립터)
메서드 `__get__()` 이나 `__set__()` 이나 `__delete__()`를 정의하는 객체. 클래스 어트리뷰트가 디스크립터일 때, 어트리뷰트 조회는 특별한 연결 작용을 일으킵니다. 보통, *a.b*를 읽거나, 쓰거나, 삭제하는데 사용할 때, *a*의 클래스 딕셔너리에서 *b* 라고 이름 붙여진 객체를 찾습니다. 하지만 *b*가 디스크립터면, 해당하는 디스크립터 메서드가 호출됩니다. 디스크립터를 이해하는 것은 파이썬에 대한 깊은 이해의 열쇠인데, 함수, 메서드, 프로퍼티, 클래스 메서드, 스태틱 메서드, 슈퍼 클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 descriptors나 디스크립터 사용법 안내서에 나옵니다.

dictionary (딕셔너리)
임의의 키를 값에 대응시키는 연관 배열 (associative array). 키는 `__hash__()` 와 `__eq__()` 메서드를 갖는 모든 객체가 될 수 있습니다. 펄에서 해시라고 부릅니다.

dictionary comprehension (딕셔너리 컴프리헨션)
이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 딕셔너리를 반환하는 간결한 방법. `results = {n: n ** 2 for n in range(10)}`은 값 `n ** 2`에 매핑된 키 `n`을 포함하는 딕셔너리를 생성합니다. comprehensions을 참조하십시오.

dictionary view (딕셔너리 뷰)
`dict.keys()`, `dict.values()`, `dict.items()` 메서드가 돌려주는 객체들을 딕셔너리 뷰라고 부릅니다. 이것들은 딕셔너리 항목들에 대한 동적인 뷰를 제공하는데, 딕셔너리가 변경될 때, 뷰가 이 변

화를 반영한다는 뜻입니다. 딕셔너리 뷰를 완전한 리스트로 바꾸려면 `list(dictview)` 를 사용하면 됩니다. dict-views를 보세요.

docstring (독스트링)
클래스, 함수, 모듈에서 첫 번째 표현식으로 나타나는 문자열 리터럴. 스위트가 실행될 때는 무시되지만, 컴파일러에 의해 인지되어 둘러싼 클래스, 함수, 모듈의 `__doc__` 어트리뷰트로 삽입됩니다. 인트로스펙션을 통해 사용할 수 있으므로, 객체의 설명서를 위한 규범적인 장소입니다.

duck-typing (덕 타이핑)
올바른 인터페이스를 가졌는지 판단하는데 객체의 형을 보지 않는 프로그래밍 스타일; 대신, 단순히 메서드나 어트리뷰트가 호출되거나 사용됩니다 ("오리처럼 보이고 오리처럼 꽥꽥댄다면, 그것은 오리다.") 특정한 형 대신에 인터페이스를 강조함으로써, 잘 설계된 코드는 다형적인 치환을 허락함으로써 유연성을 개선할 수 있습니다. 덕 타이핑은 `type()` 이나 `isinstance()` 을 사용한 검사를 피합니다. (하지만, 덕 타이핑이 추상 베이스 클래스 로 보완될 수 있음에 유의해야 합니다.) 대신에, `hasattr()` 검사나 *EAFP* 프로그래밍을 씁니다.

dunder
An informal short-hand for "double underscore", used when talking about a *special method*. For example, `__init__` is often pronounced "dunder init".

EAFP
허락보다는 용서를 구하기가 쉽다 (Easier to ask for forgiveness than permission). 이 흔히 볼 수 있는 파이썬 코딩 스타일은, 올바른 키나 어트리뷰트의 존재를 가정하고, 그 가정이 틀리면 예외를 잡습니다. 이 깔끔하고 빠른 스타일은 많은 `try` 와 `except` 문의 존재로 특징지어집니다. 이 테크닉은 C와 같은 다른 많은 언어에서 자주 사용되는 *LBYL* 스타일과 대비됩니다.

evaluate function
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression (표현식)
어떤 값으로 구해질 수 있는 문법적인 조각. 다른 말로 표현하면, 표현식은 리터럴, 이름, 어트리뷰트 액세스, 연산자, 함수들과 같은 값을 돌려주는 표현 요소들을 쌓아 올린 것입니다. 다른 많은 언어와 대조적으로, 모든 언어 구성물들이 표현식인 것은 아닙니다. `while`처럼, 표현식으로 사용할 수 없는 문장 들이 있습니다. 대입 또한 문장이고, 표현식이 아닙니다.

extension module (확장 모듈)
C 나 C++로 작성된 모듈인데, 파이썬의 C API를 사용해서 핵심이나 사용자 코드와 상호 작용합니다.

f-string (f-문자열)
f-strings
String literals prefixed with `f` or `F` are commonly called "f-strings" which is short for formatted string literals. See also **PEP 498**.

file object (파일 객체)
하부 자원에 대해 파일 지향적 API(`read()` 나 `write()` 같은 메서드들)를 드러내는 객체. 만들어진 방법에 따라, 파일 객체는 실제 디스크 상의 파일이나 다른 저장 장치나 통신 장치 (예를 들어, 표준 입출력, 인-메모리 버퍼, 소켓, 파이프, 등등)에 대한 액세스를 중계할 수 있습니다. 파일 객체는 파일류 객체 (*file-like objects*) 나 스트림 (*streams*) 이라고도 불립니다.

실제로는 세 부류의 파일 객체들이 있습니다. 날(raw) 바이너리 파일, 버퍼드(buffered) 바이너리 파일, 텍스트 파일. 이들의 인터페이스는 `io` 모듈에서 정의됩니다. 파일 객체를 만드는 규범적인 방법은 `open()` 함수를 쓰는 것입니다.

file-like object (파일류 객체)
파일 객체 의 비슷한 말.

filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기)
Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the *PyConfig_Read()* function: see *filesystem_encoding* and *filesystem_errors* members of *PyConfig*.

로케일 인코딩 도 보세요.

finder (파인더)
임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

두 종류의 파인더가 있습니다: `sys.meta_path` 와 함께 사용하는 메타 경로 파인더 와 `sys.path_hooks` 과 함께 사용하는 경로 엔트리 파인더.

더 자세한 내용은 finders-and-loaders 와 `importlib`를 참조하십시오.

floor division (정수 나눗셈)
가장 가까운 정수로 내림하는 수학적 나눗셈. 정수 나눗셈 연산자는 `//` 다. 예를 들어, 표현식 `11 // 4` 의 값은 `2`가 되지만, 실수 나눗셈은 `2.75`를 돌려줍니다. `(-11) // 4` 가 `-2.75`를 내림 한 `-3`이 됨에 유의해야 합니다. **PEP 238**을 보세요.

free threading (자유 스레딩)
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable (자유 변수)
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function (함수)
호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 인자 가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. 매개변수 와 메서드 와 function 섹션도 보세요.

function annotation (함수 어노테이션)
함수 매개변수나 반환 값의 어노테이션.

함수 어노테이션은 일반적으로 형 힌트 로 사용됩니다: 예를 들어, 이 함수는 두 개의 `int` 인자를 받아들일 것으로 기대되고, 동시에 `int` 반환 값을 줄 것으로 기대됩니다:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

함수 어노테이션 문법은 function 절에서 설명합니다.

이 기능을 설명하는 변수 어노테이션 과 **PEP 484**를 참조하세요. 또한 어노테이션에 대한 모범 사례는 annotations-howto를 참조하세요.

__future__
A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (가비지 수거)
더 사용되지 않는 메모리를 반납하는 절차. 파이썬은 참조 횟수 추적과 참조 순환을 감지하고 끊을 수 있는 순환 가비지 수거기를 통해 가비지 수거를 수행합니다. 가비지 수거기는 `gc` 모듈을 사용해서 제어할 수 있습니다.

generator (제너레이터)
제너레이터 이터레이터 를 돌려주는 함수. 일반 함수처럼 보이는데, 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다. 이 값들은 for-루프로 사용하거나 `next()` 함수로 한 번에 하나씩 꺼낼 수 있습니다.

보통 제너레이터 함수를 가리키지만, 어떤 문맥에서는 제너레이터 이터레이터 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

generator iterator (제너레이터 이터레이터)
제너레이터 함수가 만드는 객체.

각 `yield`는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 제너레이터 이터레이터 가 재개되면, 떠난 곳으로 복귀합니다 (호출마다 새로 시작하는 함수와 대비됩니다).

generator expression (제너레이터 표현식)
이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 `for` 절과 생략 가능한 `if` 절이 뒤에 붙는 일반 표현식 처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function (제네릭 함수)
같은 연산을 서로 다른 형들에 대해 구현한 여러 함수로 구성된 함수. 호출 때 어떤 구현이 사용될지는 디스패치 알고리즘에 의해 결정됩니다.

싱글 디스패치 용어집 항목과 `functools.singledispatch()` 데코레이터와 **PEP 443**도 보세요.

generic type (제네릭 형)
매개 변수화 할 수 있는 형; 일반적으로 `list` 나 `dict`와 같은 컨테이너 클래스. 형 힌트와 어노테이션에 사용됩니다.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL
전역 인터프리터 록 을 보세요.

global interpreter lock (전역 인터프리터 록)
한 번에 오직 하나의 스레드가 파이썬 바이트 코드 를 실행하도록 보장하기 위해 *CPython* 인터프리터가 사용하는 메커니즘. (`dict`와 같은 중요한 내장형들을 포함하는) 객체 모델이 묵시적으로 동시 액세스에 대해 안전하도록 만들어서 CPython 구현을 단순하게 만듭니다. 인터프리터 전체를 잠그는 것은 인터프리터를 다중스레드화하기 쉽게 만드는 대신, 다중 프로세서 기계가 제공하는 병렬성의 많은 부분을 희생합니다.

하지만, 어떤 확장 모듈들은, 표준이나 제삼자 모두, 압축이나 해싱 같은 계산 집약적인 작업을 수행할 때는 GIL을 반납하도록 설계되었습니다. 또한, I/O를 할 때는 항상 GIL을 반납합니다.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

In prior versions of Python' s C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

hash-based pyc (해시 기반 pyc)
유효성을 판별하기 위해 해당 소스 파일의 최종 수정 시간이 아닌 해시를 사용하는 바이트 코드 캐시 파일. pyc-invalidation을 참조하세요.

hashable (해시 가능)
객체가 일생 그 값이 변하지 않는 해시값을 갖고 (`__hash__()` 메서드가 필요합니다), 다른 객체와 비교될 수 있으면 (`__eq__()` 메서드가 필요합니다), 해시 가능하다고 합니다. 같다고 비교되는 해시 가능한 객체들의 해시값은 같아야 합니다.

해시 가능성은 객체를 딕셔너리의 키나 집합의 멤버로 사용할 수 있게 하는데, 이 자료 구조들이 내부적으로 해시값을 사용하기 때문입니다.

대부분 파이썬의 불변 내장 객체들은 해시 가능합니다; (리스트나 딕셔너리 같은) 가변 컨테이너들은 그렇지 않습니다; (튜플이나 frozenset 같은) 불변 컨테이너들은 그들의 요소들이 해시 가능할 때만 해시 가능합니다. 사용자 정의 클래스의 인스턴스 객체들은 기본적으로 해시 가능합니다. (자기 자신을 제외하고는) 모두 다르다고 비교되고, 해시값은 `id()`로 부터 만들어집니다.

IDLE
파이썬을 위한 통합 개발 및 학습 환경 (Integrated Development and Learning Environment). idle은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

immortal (불멸)
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via *`PyUnstable_IsImmortal()`* in the C API.

immutable (불변)
고정된 값을 갖는 객체. 불변 객체는 숫자, 문자열, 튜플을 포함합니다. 이런 객체들은 변경될 수 없습니다. 새 값을 저장하려면 새 객체를 만들어야 합니다. 변하지 않는 해시값이 있어야 하는 곳에서 중요한 역할을 합니다, 예를 들어, 딕셔너리의 키.

import path (임포트 경로)
경로 기반 파인더 가 임포트 할 모듈을 찾기 위해 검색하는 장소들 (또는 경로 엔트리) 의 목록. 임포트 하는 동안, 이 장소들의 목록은 보통 `sys.path` 로부터 옵니다, 하지만 서브 패키지의 경우 부모 패키지의 `__path__` 어트리뷰트로부터 올 수도 있습니다.

importing (임포팅)
한 모듈의 파이썬 코드가 다른 모듈의 파이썬 코드에서 사용될 수 있도록 하는 절차.

importer (임포터)
모듈을 찾기도 하고 로드 하기도 하는 객체; 동시에 파인더 이자 로더 객체입니다.

interactive (대화형)
파이썬은 대화형 인터프리터를 갖고 있는데, 인터프리터 프롬프트에서 문장과 표현식을 입력할 수 있고, 즉각 실행된 결과를 볼 수 있다는 뜻입니다. 인자 없이 단지 `python`을 실행하세요 (컴퓨터의 주메뉴에서 선택하는 것도 가능할 수 있습니다). 새 아이디어를 검사하거나 모듈과 패키지를 들여다보는 매우 강력한 방법입니다 (`help(x)` 를 기억하세요). 대화형 모드에 대한 자세한 내용은 tut-interac 를 보세요.

interpreted (인터프리티드)
바이트 코드 컴파일러의 존재 때문에 그 구분이 흐릿해지기는 하지만, 파이썬은 컴파일 언어가 아니라 인터프리터 언어입니다. 이것은 명시적으로 실행 파일을 만들지 않고도, 소스 파일을 직접 실행할 수 있다는 뜻입니다. 그 프로그램이 좀 더 천천히 실행되기는 하지만, 인터프리터 언어는 보통 컴파일 언어보다 짧은 개발/디버깅 주기를 갖습니다. 대화형 도 보세요.

interpreter shutdown (인터프리터 종료)
종료하라는 요청을 받을 때, 파이썬 인터프리터는 특별한 시기에 진입하는데, 모듈이나 여러 가지 중요한 내부 구조들과 같은 모든 할당된 자원들을 단계적으로 반납합니다. 또한, 가비지 수거기 를 여러 번 호출합니다. 사용자 정의 파괴자나 weakref 콜백에 있는 코드들의 실행을 시작시킬 수 있습니다. 종료 시기 동안 실행되는 코드는 다양한 예외들을 만날 수 있는데, 그것이 의존하는 자원들이 더 기능하지 않을 수 있기 때문입니다 (흔한 예는 라이브러리 모듈이나 경고 장치들입니다).

인터프리터 종료의 주된 원인은 실행되는 `__main__` 모듈이나 스크립트가 실행을 끝내는 것입니다.

iterable (이터러블)
멤버들을 한 번에 하나씩 돌려줄 수 있는 객체. 이터러블의 예로는 모든 (`list`, `str`, `tuple` 같은) 시퀀스 형들, `dict` 같은 몇몇 비 시퀀스 형들, 파일 객체들, `__iter__()` 나 시퀀스 개념을 구현하는 `__getitem__()` 메서드를 써서 정의한 모든 클래스의 객체들이 있습니다.

이터러블은 `for` 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 (`zip()`, `map()`, …) 에 사용될 수 있습니다. 이터러블 객체가 내장 함수 `iter()` 에 인자로 전달되면, 그 객체의 이터레이터를 돌려줍니다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효합니다. 이터러블을 사용할 때, 보통은 `iter()` 를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없습니다. `for` 문은 이것들을

여러분을 대신해서 자동으로 해주는데, 루프를 도는 동안 이터레이터를 잡아둘 이름 없는 변수를 만듭니다. 이터레이터, 시퀀스, 제너레이터 도 보세요.

iterator (이터레이터)
데이터의 스트림을 표현하는 객체. 이터레이터의 `__next__()` 메서드를 반복적으로 호출하면 (또는 내장 함수 `next()` 로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 `StopIteration` 예외를 일으킵니다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 `__next__()` 메서드 호출은 `StopIteration` 예외를 다시 일으키기만 합니다. 이터레이터는 이터레이터 객체 자신을 돌려주는 `__iter__()` 메서드를 가질 것이 요구되기 때문에, 이터레이터는 이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드입니다. (`list` 같은) 컨테이너 객체는 `iter()` 함수로 전달하거나 `for` 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

typeiter 에 더 자세한 내용이 있습니다.

CPython 구현 상세: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function (키 함수)
키 함수 또는 콜레이션(collation) 함수는 정렬(sorting)이나 배열(ordering)에 사용되는 값을 돌려주는 콜러블입니다. 예를 들어, `locale.strxfrm()` 은 로케일 특정 방식을 따르는 정렬 키를 만드는 데 사용됩니다.

파이썬의 많은 도구가 요소들이 어떻게 순서 지어지고 묶이는지를 제어하기 위해 키 함수를 받아들입니다. 이런 것들에는 `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` 이 있습니다.

키 함수를 만드는 데는 여러 방법이 있습니다. 예를 들어, `str.lower()` 메서드는 케이스 구분 없는 정렬을 위한 키 함수로 사용될 수 있습니다. 대안적으로, 키 함수는 `lambda` 표현식으로 만들 수도 있는데, 이런 식입니다: `lambda r: (r[0], r[2])`. 또한, `operator.attrgetter()`, `operator.itemgetter()`, `operator.methodcaller()` 가 세 개의 키 함수 생성자입니다. 키 함수를 만들고 사용하는 법에 대한 예로 Sorting HOW TO 를 보세요.

keyword argument (키워드 인자)
인자 를 보세요.

lambda (람다)
호출될 때 값이 구해지는 하나의 표현식 으로 구성된 이름 없는 인라인 함수. 람다 함수를 만드는 문법은 `lambda [parameters]: expression` 입니다.

LBYL
뛰기 전에 보라 (Look before you leap). 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 사전 조건들을 검사합니다. 이 스타일은 *EAFP* 접근법과 대비되고, 많은 `if` 문의 존재로 특징지어집니다.

다중 스레드 환경에서, LBYL 접근법은 "보기"와 "뛰기" 간에 경쟁 조건을 만들게 될 위험이 있습니다. 예를 들어, 코드 `if key in mapping: return mapping[key]` 는 검사 후에, 하지만 조회 전에, 다른 스레드가 *key*를 *mapping*에서 제거하면 실패할 수 있습니다. 이런 이슈는 록이나 EAFP 접근법을 사용함으로써 해결될 수 있습니다.

lexical analyzer (어휘 분석기)
Formal name for the *tokenizer*; see *token*.

list (리스트)
내장 파이썬 시퀀스. 그 이름에도 불구하고, 원소에 대한 액세스가 $O(1)$이기 때문에, 연결 리스트(linked list)보다는 다른 언어의 배열과 유사합니다.

list comprehension (리스트 컴프리헨션)
시퀀스의 요소들 전부 또는 일부를 처리하고 그 결과를 리스트로 돌려주는 간결한 방법. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 는 0에서 255 사이에 있는 짝수들의 16진수 (0x..) 들을 포함하는 문자열의 리스트를 만듭니다. `if` 절은 생략할 수 있습니다. 생략하면, `range(256)` 에 있는 모든 요소가 처리됩니다.

loader (로더)
An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

locale encoding (로케일 인코딩)
On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method (매직 메서드)
특수 메서드 의 비공식적인 비슷한 말.

mapping (매핑)
임의의 키 조회를 지원하고 `collections.abc.Mapping` 이나 `collections.abc.MutableMapping` 추상 베이스 클래스 에 지정된 메서드들을 구현하는 컨테이너 객체. 예로는 `dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` 를 들 수 있습니다.

meta path finder (메타 경로 파인더)
`sys.meta_path` 의 검색이 돌려주는 파인더. 메타 경로 파인더는 경로 엔트리 파인더 와 관련되어 있기는 하지만 다릅니다.

메타 경로 파인더가 구현하는 메서드들에 대해서는 `importlib.abc.MetaPathFinder` 를 보면 됩니다.

metaclass (메타 클래스)
클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 딕셔너리, 베이스 클래스들의 목록을 만듭니다. 메타 클래스는 이 세 인자를 받아서 클래스를 만드는 책임을 집니다. 대부분의 객체 지향형 프로그래밍 언어들은 기본 구현을 제공합니다. 파이썬을 특별하게 만드는 것은 커스텀 메타 클래스를 만들 수 있다는 것입니다. 대부분 사용자에게는 이 도구가 전혀 필요 없지만, 필요가 생길 때, 메타 클래스는 강력하고 우아한 해법을 제공합니다. 어트리뷰트 액세스의 로깅(logging), 스레드 안전성의 추가, 객체 생성 추적, 싱글톤 구현과 많은 다른 작업에 사용됐습니다.

metaclasses 에서 더 자세한 내용을 찾을 수 있습니다.

method (메서드)
클래스 바디 안에서 정의되는 함수. 그 클래스의 인스턴스의 어트리뷰트로서 호출되면, 그 메서드는 첫 번째 인자 (보통 `self` 라고 불린다) 로 인스턴스 객체를 받습니다. 함수 와 중첩된 스코프 를 보세요.

method resolution order (메서드 결정 순서)
메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서입니다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 python_2.3_mro를 보세요.

module (모듈)
파이썬 코드의 조직화 단위를 담당하는 객체. 모듈은 임의의 파이썬 객체들을 담는 이름 공간을 갖습니다. 모듈은 임포팅 절차에 의해 파이썬으로 로드됩니다.

패키지 도 보세요.

module spec (모듈 스펙)
모듈을 로드하는데 사용되는 임포트 관련 정보들을 담고 있는 이름 공간. `importlib.machinery.ModuleSpec` 의 인스턴스.

module-specs 도 보세요.

MRO
메서드 결정 순서 를 보세요.

mutable (가변)
가변 객체는 값이 변할 수 있지만 `id()` 는 일정하게 유지합니다. 불변 도 보세요.

named tuple (네임드 튜플)
"named tuple(네임드 튜플)" 이라는 용어는 튜플에서 상속하고 이름 붙은 어트리뷰트를 사용하여 인덱스 할 수 있는 요소에 액세스 할 수 있는 모든 형이나 클래스에 적용됩니다. 형이나 클래스에는 다른 기능도 있을 수 있습니다.

`time.localtime()` 과 `os.stat()` 가 반환한 값을 포함하여, 여러 내장형이 네임드 튜플입니다. 또 다른 예는 `sys.float_info` 입니다:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

일부 네임드 튜플은 내장형(위의 예)입니다. 또는, `tuple`에서 상속하고 이름 붙은 필드를 정의하는 일반 클래스 정의로 네임드 튜플을 만들 수 있습니다. 이러한 클래스는 직접 작성하거나, `typing.NamedTuple`를 계승하거나 팩토리 함수 `collections.namedtuple()`로 만들 수 있습니다. 후자의 기법은 직접 작성하거나 내장 네임드 튜플에서는 찾을 수 없는 몇 가지 추가 메서드를 추가하기도 합니다.

namespace (이름 공간)
변수가 저장되는 장소. 이름 공간은 딕셔너리로 구현됩니다. 객체에 중첩된 이름 공간(메서드 에서) 뿐만 아니라 지역, 전역, 내장 이름 공간이 있습니다. 이름 공간은 이름 충돌을 방지해서 모듈성을 지원합니다. 예를 들어, 함수 `builtins.open` 과 `os.open()` 은 그들의 이름 공간에 의해 구별됩니다. 또한, 이름 공간은 어떤 모듈이 함수를 구현하는지를 분명하게 만들어서 가독성과 유지 보수성에 도움을 줍니다. 예를 들어, `random.seed()` 또는 `itertools.islice()` 라고 쓰면 그 함수들이 각각 `random` 과 `itertools` 모듈에 의해 구현되었음이 명확해집니다.

namespace package (이름 공간 패키지)
오직 서브 패키지들의 컨테이너로만 기능하는 패키지. 이름 공간 패키지는 물리적인 실체가 없을 수도 있고, 특히 `__init__.py` 파일이 없으므로 정규 패키지 와는 다릅니다.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

모듈 도 보세요.

nested scope (중첩된 스코프)
둘러싼 정의에서 변수를 참조하는 능력. 예를 들어, 다른 함수 내부에서 정의된 함수는 바깥 함수에 있는 변수들을 참조할 수 있습니다. 중첩된 스코프는 기본적으로는 참조만 가능할 뿐, 대입은 되지 않는다는 것에 주의해야 합니다. 지역 변수들은 가장 내부의 스코프에서 읽고 씁니다. 마찬가지로, 전역 변수들은 전역 이름 공간에서 읽고 씁니다. `nonlocal` 은 바깥 스코프에 쓰는 것을 허락합니다.

new-style class (뉴스타일 클래스)
지금은 모든 클래스 객체에 사용되고 있는 클래스 버전의 예전 이름. 초기의 파이썬 버전에서는, 오직 뉴스타일 클래스만 `__slots__`, 디스크립터, 프라퍼티, `__getattribute__()`, 클래스 메서드, 스태틱 메서드와 같은 파이썬의 새롭고 다양한 기능들을 사용할 수 있었습니다.

object (객체)
상태 (어트리뷰트나 값) 를 갖고 동작 (메서드) 이 정의된 모든 데이터. 또한, 모든 뉴스타일 클래스의 최종적인 베이스 클래스입니다.

optimized scope (최적화된 스코프)
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators,

coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package (패키지)
서브 모듈들이나, 재귀적으로 서브 패키지들을 포함할 수 있는 파이썬 모듈. 기술적으로, 패키지는 `__path__` 어트리뷰트가 있는 파이썬 모듈입니다.

정규 패키지 와 이름 공간 패키지 도 보세요.

parameter (매개변수)
함수 (또는 메서드) 정의에서 함수가 받을 수 있는 인자 (또는 어떤 경우 인자들) 를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있습니다:

- 위치-키워드 (*positional-or-keyword*): 위치 인자 나 키워드 인자 로 전달될 수 있는 인자를 지정합니다. 이것이 기본 형태의 매개변수입니다, 예를 들어 다음에서 *foo* 와 *bar*:

```
def func(foo, bar=None): ...
```

- 위치-전용 (*positional-only*): 위치로만 제공될 수 있는 인자를 지정합니다. 위치 전용 매개변수는 함수 정의의 매개변수 목록에 / 문자를 포함하고 그 뒤에 정의할 수 있습니다, 예를 들어 다음에서 *posonly1* 과 *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- 키워드-전용 (*keyword-only*): 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 `*`를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 *kw_only1* 와 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- 가변-위치 (*var-positional*): (다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정합니다. 이런 매개변수는 매개변수 이름에 `*` 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 다음에서 *args*:

```
def func(*args, **kwargs): ...
```

- 가변-키워드 (*var-keyword*): (다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정합니다. 이런 매개변수는 매개변수 이름에 `**`를 앞에 붙여서 정의될 수 있습니다, 예를 들어 위의 예에서 *kwargs*.

매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있습니다.

인자 용어집 항목, 인자와 매개변수의 차이에 나오는 FAQ 질문, `inspect.Parameter` 클래스, function 절, **PEP 362**도 보세요.

path entry (경로 엔트리)
경로 기반 파인더 가 임포트 할 모듈들을 찾기 위해 참고하는 임포트 경로 상의 하나의 장소.

path entry finder (경로 엔트리 파인더)
`sys.path_hooks` 에 있는 콜러블 (즉, 경로 엔트리 훅) 이 돌려주는 파인더 인데, 주어진 경로 엔트리 로 모듈을 찾는 방법을 알고 있습니다.

경로 엔트리 파인더들이 구현하는 메서드들은 `importlib.abc.PathEntryFinder` 에 나옵니다.

path entry hook (경로 엔트리 훅)
`sys.path_hooks` 리스트에 있는 콜러블인데, 특정 경로 엔트리 에서 모듈을 찾는 법을 알고 있다면 경로 엔트리 파인더 를 돌려줍니다.

path based finder (경로 기반 파인더)
기본 메타 경로 파인더들 중 하나인데, 임포트 경로 에서 모듈을 찾습니다.

path-like object (경로류 객체)
파일 시스템 경로를 나타내는 객체. 경로류 객체는 경로를 나타내는 `str` 나 `bytes` 객체이거

나 os.PathLike 프로토콜을 구현하는 객체입니다. os.PathLike 프로토콜을 지원하는 객체는 os.fspath() 함수를 호출해서 str 나 bytes 파일 시스템 경로로 변환될 수 있습니다; 대신 os.fsdecode() 와 os.fsencode() 는 각각 str 나 bytes 결과를 보장하는데 사용될 수 있습니다. **PEP 519**로 도입되었습니다.

PEP
파이썬 개선 제안. PEP는 파이썬 커뮤니티에 정보를 제공하거나 파이썬 또는 그 프로세스 또는 환경에 대한 새로운 기능을 설명하는 설계 문서입니다. PEP는 제안된 기능에 대한 간결한 기술 사양 및 근거를 제공해야 합니다.

PEP는 주요 새로운 기능을 제안하고 문제에 대한 커뮤니티 입력을 수집하며 파이썬에 들어간 설계 결정을 문서로 만들기 위한 기본 메커니즘입니다. PEP 작성자는 커뮤니티 내에서 합의를 구축하고 반대 의견을 문서화 할 책임이 있습니다.

PEP 1 참조하세요.

portion (포션)
PEP 420 에서 정의한 것처럼, 이름 공간 패키지에 이바지하는 하나의 디렉터리에 들어있는 파일들의 집합 (zip 파일에 저장되는 것도 가능합니다).

positional argument (위치 인자)
인자 를 보세요.

provisional API (잠정 API)
잠정 API는 표준 라이브러리의 과거 호환성 보장으로부터 신중히 제외된 것입니다. 인터페이스의 큰 변화가 예상되지는 않지만, 잠정적이라고 표시되는 한, 코어 개발자들이 필요하다고 생각한다면 과거 호환성이 유지되지 않는 변경이 일어날 수 있습니다. 그런 변경은 불필요한 방식으로 일어나지는 않을 것입니다 — API를 포함하기 전에 놓친 중대하고 근본적인 결함이 발견된 경우에만 일어날 것입니다.

잠정 API에서조차도, 과거 호환성이 유지되지 않는 변경은 "최후의 수단"으로 여겨집니다 - 모든 식별된 문제들에 대해 과거 호환성을 유지하는 해법을 찾으려는 모든 시도가 선행됩니다.

이 절차는 표준 라이브러리가 오랜 시간 동안 잘못된 설계 오류에 발목 잡히지 않고 발전할 수 있도록 만듭니다. 더 자세한 내용은 **PEP 411**을 보면 됩니다.

provisional package (잠정 패키지)
잠정 *API* 를 보세요.

Python 3000 (파이썬 3000)
파이썬 3.x 배포 라인의 별명 (버전 3의 배포가 먼 미래의 이야기던 시절에 만들어진 이름이다.) 이것을 "Py3k" 로 줄여 쓰기도 합니다.

Pythonic (파이썬다운)
다른 언어들에서 일반적인 개념들을 사용해서 코드를 구현하는 대신, 파이썬 언어에서 가장 자주 사용되는 이디엄들을 가까이 따르는 아이디어나 코드 조각. 예를 들어, 파이썬에서 자주 쓰는 이디엄은 for 문을 사용해서 이터러블의 모든 요소로 루핑하는 것입니다. 다른 많은 언어에는 이런 종류의 구성물이 없으므로, 파이썬에 익숙하지 않은 사람들은 대신에 숫자 카운터를 사용하기도 합니다:

```
for i in range(len(food)):
    print(food[i])
```

더 깔끔한, 파이썬다운 방법은 이렇습니다:

```
for piece in food:
    print(piece)
```

qualified name (정규화된 이름)
모듈의 전역 스코프에서 모듈에 정의된 클래스, 함수, 메서드에 이르는 "경로"를 보여주는 점으로 구분된 이름. **PEP 3155** 에서 정의됩니다. 최상위 함수와 클래스의 경우에, 정규화된 이름은 객체의 이름과 같습니다:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

모듈을 가리키는데 사용될 때, 완전히 정규화된 이름(*fully qualified name*)은 모든 부모 패키지들을 포함해서 모듈로 가는 점으로 분리된 이름을 의미합니다, 예를 들어, `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count (참조 횟수)
객체에 대한 참조의 개수. 객체의 참조 횟수가 0으로 떨어지면, 메모리가 반납됩니다. 일부 객체는 불멸이며 참조 횟수가 수정되지 않아서, 객체가 할당 해제되지 않습니다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소입니다. 프로그래머는 특정 객체의 참조 횟수를 돌려주는 `sys.getrefcount()` 함수를 호출할 수 있습니다.

In *CPython*, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

regular package (정규 패키지)
`__init__.py` 파일을 포함하는 디렉터리와 같은 전통적인 패키지.

이름 공간 패키지 도 보세요.

REPL
An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__
클래스 내부의 선언인데, 인스턴스 어트리뷰트들을 위한 공간을 미리 선언하고 인스턴스 딕셔너리를 제거함으로써 메모리를 절감하는 효과를 줍니다. 인기 있기는 하지만, 이 테크닉은 올바르게 사용하기가 좀 까다로운 편이라서, 메모리에 민감한 응용 프로그램에서 많은 수의 인스턴스가 있는 특별한 경우로 한정하는 것이 좋습니다.

sequence (시퀀스)
`__getitem__()` 특수 메서드를 통해 정수 인덱스를 사용한 빠른 요소 액세스를 지원하고, 시퀀스의 길이를 돌려주는 `__len__()` 메서드를 정의하는 이터러블. 몇몇 내장 시퀀스들을 나열해보면, `list`, `str`, `tuple`, `bytes` 가 있습니다. `dict` 또한 `__getitem__()` 과 `__len__()` 을 지원하지만, 조회에 정수 대신 임의의 해시 가능 키를 사용하기 때문에 시퀀스가 아니라 매핑으로 취급된다는 것에 주의해야 합니다.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension (집합 컴프리헨션)
이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 집합을 반환하는 간결한 방법. `results = {c for c in 'abracadabra' if c not in 'abc'}`는 문자열의 집합 `{'r', 'd'}`를 생성합니다. comprehensions을 참조하십시오.

single dispatch (싱글 디스패치)
구현이 하나의 인자의 형에 기초해서 결정되는 제네릭 함수 디스패치의 한 형태.

slice (슬라이스)
보통 시퀀스 의 일부를 포함하는 객체. 슬라이스는 서브 스크립트 표기법을 사용해서 만듭니다. `variable_name[1:3:5]` 처럼, `[]` 안에서 여러 개의 숫자를 콜론으로 분리합니다. 대괄호 (서브 스크립트) 표기법은 내부적으로 `slice` 객체를 사용합니다.

soft deprecated (약하게 폐지된)
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

special method (특수 메서드)
파이썬이 형에 어떤 연산을, 덧셈 같은, 실행할 때 묵시적으로 호출되는 메서드. 이런 메서드는 두 개의 밑줄로 시작하고 끝나는 이름을 갖고 있습니다. 특수 메서드는 specialnames 에 문서로 만들어져 있습니다.

standard library
The collection of *packages*, *modules* and *extension modules* distributed as a part of the official Python interpreter package. The exact membership of the collection may vary based on platform, available system libraries, or other criteria. Documentation can be found at library-index.

See also `sys.stdlib_module_names` for a list of all possible standard library module names.

statement (문장)
문장은 스위트 (코드의 "블록(block)") 를 구성하는 부분입니다. 문장은 표현식 이거나 키워드를 사용하는 여러 가지 구조물 중의 하나입니다. 가령 `if`, `while`, `for`.

static type checker (정적 형 검사기)
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

stdlib
An abbreviation of *standard library*.

strong reference (강한 참조)
In Python' s C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling *`Py_INCREF()`* when the reference is created and released with *`Py_DECREF()`* when the reference is deleted.

The *`Py_NewRef()`* function can be used to create a strong reference to an object. Usually, the *`Py_DECREF()`* function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

빌린 참조 도 보세요.

t-string
t-strings
String literals prefixed with `t` or `T` are commonly called "t-strings" which is short for template string literals.

text encoding (텍스트 인코딩)
A string in Python is a sequence of Unicode code points (in range `U+0000–U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization codecs, which are collectively referred to as "text encodings".

text file (텍스트 파일)
`str` 객체를 읽고 쓸 수 있는 파일 객체. 종종, 텍스트 파일은 실제로는 바이트 지향 데이터스트림을 액세스하고 텍스트 인코딩 을 자동 처리합니다. 텍스트 파일의 예로는 텍스트 모드 (`'r'` 또는 `'w'`) 로 열린 파일, `sys.stdin`, `sys.stdout`, `io.StringIO` 의 인스턴스를 들 수 있습니다.

바이트열류 객체 를 읽고 쓸 수 있는 파일 객체에 대해서는 바이너리 파일 도 참조하세요.

thread state
The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python' s C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See *Thread State and the Global Interpreter Lock* for more information.

token (토큰)
A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python' s lexical analyzer. The `token` module contains information on the various types of tokens.

triple-quoted string (삼중 따옴표 된 문자열)
따옴표 (") 나 작은따옴표 (') 세 개로 둘러싸인 문자열. 그냥 따옴표 하나로 둘러싸인 문자열에 없는 기능을 제공하지는 않지만, 여러 가지 이유에서 쓸모가 있습니다. 이스케이프 되지 않은 작은따옴표나 큰따옴표를 문자열 안에 포함할 수 있도록 하고, 연결 문자를 쓰지 않고도 여러 줄에 걸칠 수 있는데, 독스트링을 쓸 때 특히 쓸모 있습니다.

type (형)
파이썬 객체의 형은 그것이 어떤 종류의 객체인지를 결정합니다; 모든 객체는 형이 있습니다. 객체의 형은 `__class__` 어트리뷰트로 액세스할 수 있거나 `type(obj)` 로 얻을 수 있습니다.

type alias (형 에일리어스)
형을 식별자에 대입하여 만들어지는 형의 동의어.

형 에일리어스는 형 힌트를 단순화하는 데 유용합니다. 예를 들면:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

이 기능을 설명하는 `typing`과 **PEP 484**를 참조하세요.

type hint (형 힌트)
변수, 클래스 어트리뷰트 및 함수 매개변수 나 반환 값의 기대되는 형을 지정하는 어노테이션.

형 힌트는 선택 사항이고 파이썬에서 강제되지는 않지만, 정적 형 검사기에 유용합니다. 또한 IDE의 코드 완성 및 리팩토링을 돕습니다.

지역 변수를 제외하고, 전역 변수, 클래스 어트리뷰트 및 함수의 형 힌트는 `typing.get_type_hints()`를 사용하여 액세스할 수 있습니다.

이 기능을 설명하는 `typing`과 **PEP 484**를 참조하세요.

universal newlines (유니버설 줄 넘김)
다음과 같은 것들을 모두 줄의 끝으로 인식하는, 텍스트 스트림을 해석하는 태도: 유닉스 개행 문자 관례 `'\n'`, 윈도우즈 관례 `'\r\n'`, 예전의 매킨토시 관례 `'\r'`. 추가적인 사용에 관해서는 `bytes.splitlines()` 뿐만 아니라 **PEP 278** 와 **PEP 3116** 도 보세요.

variable annotation (변수 어노테이션)
변수 또는 클래스 어트리뷰트의 어노테이션.

변수 또는 클래스 어트리뷰트에 어노테이션을 달 때 대입은 선택 사항입니다:

```
class C:
    field: 'annotation'
```

변수 어노테이션은 일반적으로 형 힌트로 사용됩니다: 예를 들어, 이 변수는 `int` 값을 가질 것으로 기대됩니다:

```
count: int = 0
```

변수 어노테이션 문법은 섹션 annassign 에서 설명합니다.

이 기능을 설명하는 함수 어노테이션, **PEP 484** 및 **PEP 526**을 참조하세요. 또한 어노테이션 작업에 대한 모범 사례는 annotations-howto를 참조하세요.

virtual environment (가상 환경)
파이썬 사용자와 응용 프로그램이, 같은 시스템에서 실행되는 다른 파이썬 응용 프로그램들의 동작에 영향을 주지 않으면서, 파이썬 배포 패키지들을 설치하거나 업그레이드하는 것을 가능하게 하는, 협력적으로 격리된 실행 환경.

`venv` 도 보세요.

virtual machine (가상 기계)
소프트웨어만으로 정의된 컴퓨터. 파이썬의 가상 기계는 바이트 코드 컴파일러가 출력하는 바이트 코드를 실행합니다.

walrus operator
A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Zen of Python (파이썬 젠)
파이썬 디자인 원리와 철학들의 목록인데, 언어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 "`import this`" 를 입력하면 보입니다.

APPENDIX B

이 설명서에 관하여

파이썬 설명서는 reStructuredText 소스에서 만들어진 것으로, 원래 파이썬을 위해 제작되었고 이제는 독립 프로젝트로 유지 관리되는 설명서 생성기인 Sphinx 를 사용했습니다.

설명서와 이를 위한 툴체인 개발은 파이썬 자체와 마찬가지로 전적으로 자원봉사자의 노력입니다. 기여하고 싶다면, 참여 방법에 대한 정보는 reporting-bugs 페이지를 참고하십시오. 새로운 자원봉사자는 언제나 환영합니다!

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 저자;
- reStructuredText와 Docutils 스위트를 만드는 Docutils 프로젝트.
- Fredrik Lundh, 그의 대안 파이썬 참조(Alternative Python Reference) 프로젝트에서 Sphinx가 많은 아이디어를 얻었습니다.

B.1 파이썬 설명서의 공헌자들

많은 사람이 파이썬 언어, 파이썬 표준 라이브러리 및 파이썬 설명서에 기여했습니다. 기여자의 부분적인 목록은 파이썬 소스 배포판의 Misc/ACKS 를 참조하십시오.

파이썬이 이런 멋진 설명서를 갖게 된 것은 파이썬 커뮤니티의 입력과 기여 때문입니다 – 감사합니다!

APPENDIX C

역사와 라이센스

C.1 소프트웨어의 역사

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python' s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

배포판	파생된 곳	해	소유자	GPL-compatible? (1)
0.9.0 ~ 1.2	n/a	1991-1995	CWI	yes
1.3 ~ 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 이상	2.1.1	2001-현재	PSF	yes

참고

(1) GPL-compatible doesn' t mean that we' re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-

compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don' t.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman' s lawyer has told CNRI' s lawyer that 1.6.1 is "not incompatible" with the GPL.

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

파이썬에 통합된 일부 소프트웨어에는 다른 라이센스가 적용됩니다. 라이센스는 해당 라이센스에 해당하는 코드와 함께 나열됩니다. 이러한 라이센스의 불완전한 목록은 포함된 소프트웨어에 대한 라이센스 및 승인를 참조하십시오.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 포함된 소프트웨어에 대한 라이센스 및 승인

이 섹션은 파이썬 배포판에 포함된 제삼자 소프트웨어에 대한 불완전하지만 늘어나고 있는 라이센스와 승인의 목록입니다.

C.3.1 메르센 트위스터

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 소켓

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 비동기 소켓 서비스

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 쿠키 관리

`http.cookies` 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 실행 추적

`trace` 모듈은 다음과 같은 주의 사항을 포함합니다:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 및 UUdecode 함수

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 원격 프로시저 호출

xmlrpc.client 모듈은 다음과 같은 주의 사항을 포함합니다:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The test.test_epoll module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

파일 `Python/pyhash.c` 에는 Dan Bernstein의 SipHash24 알고리즘의 Marek Majkowski의 구현이 포함되어 있습니다. 여기에는 다음과 같은 내용이 포함되어 있습니다:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 와 dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

`zlib` 확장은 시스템에서 발견된 zlib 버전이 너무 오래되어서 빌드에 사용될 수 없으면, 포함된 zlib 소스 사본을 사용하여 빌드됩니다:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

`tracemalloc` 에 의해 사용되는 해시 테이블의 구현은 cfuhash 프로젝트를 기반으로 합니다:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 테스트 스위트

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD' s "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.22 Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library, copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License:

```
Copyright (c) 2020-present, Ma Lin and contributors.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX D

저작권

파이썬과 이 설명서는:

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

전체 라이센스 및 사용 권한 정보는 역사와 라이센스 에서 제공합니다.

Bibliography

[win] `PyExc_WindowsError` is only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

색인

알파벳 이외

... , **369**
    ellipsis literal, 369
>>>, **369**
__all__ (패키지 변수), 82
__dict__ (모듈 어트리뷰트), 203
__doc__ (모듈 어트리뷰트), 203
__file__ (모듈 어트리뷰트), 203, 204
__future__, **375**
__import__
    내장 함수, 83
__loader__ (모듈 어트리뷰트), 203
__main__
    module, 233, 248, 249
    모듈, 12
__name__ (모듈 어트리뷰트), 203
__package__ (모듈 어트리뷰트), 203
__PYVENV_LAUNCHER__, 272, 278
__slots__, **383**
_frozen (*C struct*), 85
_inittab (*C struct*), 86
_inittab.initfunc (*C member*), 86
_inittab.name (*C member*), 86
_Py_c_diff (*C function*), 155
_Py_c_neg (*C function*), 155
_Py_c_pow (*C function*), 155
_Py_c_prod (*C function*), 155
_Py_c_quot (*C function*), 155
_Py_c_sum (*C function*), 154
_Py_NoneStruct (*C var*), 301
_PyBytes_Resize (*C function*), 158
_PyCode_GetExtra (*C* 함수), 201
_PyCode_SetExtra (*C* 함수), 201
_PyEval_RequestCodeExtraIndex (*C* 함수), 201
_PyFrameEvalFunction (*C type*), 246
_PyInterpreterFrame (*C struct*), 219
_PyInterpreterState_GetEvalFrameFunc (*C function*), 246
_PyInterpreterState_SetEvalFrameFunc (*C function*), 246
_PyObject_GetDictPtr (*C function*), 109
_PyObject_New (*C function*), 299
_PyObject_NewVar (*C function*), 299
_PyTuple_Resize (*C function*), 182
_thread
    module, 242
객체
    frozenset, 190
    long 정수, 142
    memoryview, 213
    None, 142
    set, 190
    딕셔너리, 185
    리스트, 183
    메서드, 195
    모듈, 203
    바이트 배열, 158
    바이트열, 156
    복소수, 154
    부동 소수점, 152
    인스턴스 메서드, 194
    정수, 142
    캡슐, 216
    코드, 196
    튜플, 181
    파일, 202
    함수, 192
    형, 6, 135
검색
    경로, 모듈, 12
경로
    모듈 검색, 12
시그널
    모듈, 61
아스키
    내장 함수, 109
연속, 130
인스턴스 메서드
    객체, 194
정리 함수, 82
정수
    객체, 142
캡슐
    객체, 216
컴파일
    내장 함수, 84
코드 객체, 196

A

abort (*C* 함수), 82
abs
    내장 함수, 120
abstract base class (추상 베이스 클래스), **369**
allocfunc (*C type*), 350
annotate function, **369**
annotation (어노테이션), **369**
argument (인자), **369**
argv (*in module sys*), 238, 271
asynchronous context manager (비동기 컨텍스트 관리자), **370**
asynchronous generator (비동기 제너레이터), **370**
asynchronous generator iterator (비동기 제너레이터 이터레이터), **370**
asynchronous iterable (비동기 이터러블), **370**
asynchronous iterator (비동기 이터레이터), **370**
attached thread state, **370**
attribute (어트리뷰트), **371**
awaitable (어웨이터블), **371**

B

BDFL, **371**
binary file (바이너리 파일), **371**
binaryfunc (*C type*), 351
borrowed reference (빌린 참조), **371**
built-in function
    hash, 326
    repr, 325
builtins
    module, 233, 248, 249
    모듈, 12
bytecode (바이트 코드), **371**
bytes-like object (바이트열류 객체), **371**

C

callable (콜러블), **371**
callback (콜백), **372**
calloc (*C* 함수), 287
C-contiguous, 372
class (클래스), **372**
class variable (클래스 변수), **372**
classmethod
    내장 함수, 309
close (*in module os*), 249
closure variable (클로저 변수), **372**
CO_ASYNC_GENERATOR (*C macro*), 200
CO_COROUTINE (*C macro*), 200
CO_FUTURE_ABSOLUTE_IMPORT (*C macro*), 200
CO_FUTURE_ANNOTATIONS (*C macro*), 200
CO_FUTURE_DIVISION (*C macro*), 200
CO_FUTURE_GENERATOR_STOP (*C macro*), 200
CO_FUTURE_PRINT_FUNCTION (*C macro*), 200
CO_FUTURE_UNICODE_LITERALS (*C macro*), 200
CO_FUTURE_WITH_STATEMENT (*C macro*), 200
CO_GENERATOR (*C macro*), 200
CO_HAS_DOCSTRING (*C macro*), 200
CO_ITERABLE_COROUTINE (*C macro*), 200
CO_METHOD (*C macro*), 200
CO_NESTED (*C macro*), 200
CO_NEWLOCALS (*C macro*), 200
CO_OPTIMIZED (*C macro*), 200
CO_VARARGS (*C macro*), 200
CO_VARKEYWORDS (*C macro*), 200
Common Vulnerabilities and Exposures
    CVE 2008-5983, 238
complex number (복소수), **372**
context (컨텍스트), **372**
context management protocol (컨텍스트 관리 프로토콜), **372**
context manager (컨텍스트 관리자), **372**
context variable (컨텍스트 변수), **372**
contiguous (연속), **372**
copyright (*in module sys*), 238
coroutine (코루틴), **373**
coroutine function (코루틴 함수), **373**
CPython, **373**
current context (현재 컨텍스트), **373**
C-연속, 130
cyclic isolate, **373**

D

decorator (데코레이터), **373**
descrgetfunc (*C type*), 351
descriptor (디스크립터), **373**
descrsetfunc (*C type*), 351
destructor (*C type*), 351
dictionary (딕셔너리), **373**
dictionary comprehension (딕셔너리 컴프리헨션), **373**
dictionary view (딕셔너리 뷰), **373**
divmod
    내장 함수, 120
docstring (독스트링), **374**
duck-typing (덕 타이핑), **374**
dunder, **374**

E

EAFP, **374**
EOFError (내장 예외), 202
evaluate function, **374**
exc_info (모듈 *sys* 에서), 10
executable (*in module sys*), 237
exit (*C* 함수), 82
expression (표현식), **374**
extension module (확장 모듈), **374**

F

f-string (*f*-문자열), **374**
f-strings, **374**
file object (파일 객체), **374**
file-like object (파일류 객체), **374**
filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기), **374**

finder (파인더), **375**
float
    내장 함수, 122
floor division (정수 나눗셈), **375**
Fortran contiguous, 372
free (*C* 함수), 287
free threading (자유 스레딩), **375**
free variable (자유 변수), **375**
freefunc (*C type*), 351
freeze 유틸리티, 85
frozenset
    객체, 190
function (함수), **375**
function annotation (함수 어노테이션), **375**

G

garbage collection (가비지 수거), **375**
gcvisitobjects_t (*C type*), 358
generator (제너레이터), 375, **376**
generator expression (제너레이터 표현식), **376**
generator iterator (제너레이터 이터레이터), **376**
generic function (제네릭 함수), **376**
generic type (제네릭 형), **376**
getattrfunc (*C type*), 351
getattrofunc (*C type*), 351
getbufferproc (*C type*), 351
getiterfunc (*C type*), 351
getter (*C type*), 314
GIL, **376**
global interpreter lock, 239
global interpreter lock (전역 인터프리터 록), **376**

H

hash
    built-in function, 326
hash-based pyc (해시 기반 *pyc*), **376**
hashable (해시 가능), **376**
hashfunc (*C type*), 351

I

IDLE, **377**
immortal (불멸), **377**
immutable (불변), **377**
import path (임포트 경로), **377**
importer (임포터), **377**
importing (임포팅), **377**
incr_item(), 11
initproc (*C type*), 351
inquiry (*C type*), 357
int
    내장 함수, 122
interactive (대화형), **377**
interpreted (인터프리티드), **377**
interpreter lock, 239
interpreter shutdown (인터프리터 종료), **377**
iterable (이터러블), **377**
iterator (이터레이터), **378**
iternextfunc (*C type*), 351

K

key function (키 함수), **378**
KeyboardInterrupt (내장 예외), 61
keyword argument (키워드 인자), **378**

L

lambda (람다), **378**
LBYL, **378**
len
    내장 함수, 111, 123, 125, 184, 188, 191
lenfunc (*C type*), 351
lexical analyzer (어휘 분석기), **378**
list (리스트), **378**
list comprehension (리스트 컴프리헨션), **378**
loader (로더), **379**
locale encoding (로케일 인코딩), **379**
lock, interpreter, 239
long 정수
    객체, 142
LONG_MAX (*C* 매크로), 144

M

magic
    method (메서드), 379
magic method (매직 메서드), **379**
main(), 236, 238, 271
malloc (*C* 함수), 287
mapping
    object, 185
mapping (매핑), **379**
memoryview
    객체, 213
meta path finder (메타 경로 파인더), **379**
metaclass (메타 클래스), **379**
METH_CLASS (*C macro*), 309
METH_COEXIST (*C macro*), 309
METH_FASTCALL (*C macro*), 309
METH_KEYWORDS (*C macro*), 308
METH_METHOD (*C macro*), 309
METH_NOARGS (*C macro*), 309
METH_O (*C macro*), 309
METH_STATIC (*C macro*), 309
METH_VARARGS (*C macro*), 308
method (메서드), **379**
    magic, 379
    special, 384
method resolution order (메서드 결정 순서), **379**
MethodType (*types* 모듈), 192, 195
module
    __main__, 233, 248, 249
    _thread, 242
    builtins, 233, 248, 249
    search path, 233, 237
    sys, 233, 248, 249

module (모듈), **379**
module spec (모듈 스펙), **379**
modules (*in module sys*), 233
modules (*sys* 모듈), 82
ModuleType (모듈 *types*), 203
MRO, **380**
mutable (가변), **380**

N

named tuple (네임드 튜플), **380**
namespace (이름 공간), **380**
namespace package (이름 공간 패키지), **380**
nested scope (중첩된 스코프), **380**
new-style class (뉴스타일 클래스), **380**
newfunc (*C type*), 351
None
    객체, 142
numeric
    object, 142

O

object
    mapping, 185
    numeric, 142
    sequence, 156
object (객체), **380**
objobjargproc (*C type*), 352
objobjproc (*C type*), 352
optimized scope (최적화된 스코프), **380**
OverflowError (내장 예외), 144, 145

P

package (패키지), **381**
parameter (매개변수), **381**
PATH, 12
path
    module search, 233, 237
path (*in module sys*), 233, 237
path (모듈 *sys* 에서), 12
path based finder (경로 기반 파인더), **381**
path entry (경로 엔트리), **381**
path entry finder (경로 엔트리 파인더), **381**
path entry hook (경로 엔트리 훅), **381**
path-like object (경로류 객체), **381**
PEP, **382**
platform (*in module sys*), 237
portion (포션), **382**
positional argument (위치 인자), **382**
pow
    내장 함수, 120, 122
provisional API (잠정 *API*), **382**
provisional package (잠정 패키지), **382**
Py_ABS (*C macro*), 4
Py_AddPendingCall (*C function*), 250
Py_ALWAYS_INLINE (*C macro*), 4
Py_ASNATIVEBYTES_ALLOW_INDEX (*C macro*), 148
Py_ASNATIVEBYTES_BIG_ENDIAN (*C macro*), 148
Py_ASNATIVEBYTES_DEFAULTS (*C macro*), 148
Py_ASNATIVEBYTES_LITTLE_ENDIAN (*C macro*), 148
Py_ASNATIVEBYTES_NATIVE_ENDIAN (*C macro*), 148
Py_ASNATIVEBYTES_REJECT_NEGATIVE (*C macro*), 148
Py_ASNATIVEBYTES_UNSIGNED_BUFFER (*C macro*), 148
Py_AtExit (*C function*), 82
Py_AUDIT_READ (*C macro*), 311
Py_AuditHookFunction (*C type*), 82
Py_BEGIN_ALLOW_THREADS (*C macro*), 240, 244
Py_BEGIN_CRITICAL_SECTION (*C macro*), 257
Py_BEGIN_CRITICAL_SECTION2 (*C macro*), 258
Py_BEGIN_CRITICAL_SECTION2_MUTEX (*C macro*), 258
Py_BEGIN_CRITICAL_SECTION_MUTEX (*C macro*), 257
Py_BLOCK_THREADS (*C macro*), 244
Py_buffer (*C type*), 128
Py_buffer.buf (*C member*), 128
Py_buffer.format (*C member*), 128
Py_buffer.internal (*C member*), 129
Py_buffer.itemsize (*C member*), 128
Py_buffer.len (*C member*), 128
Py_buffer.ndim (*C member*), 128
Py_buffer.obj (*C member*), 128
Py_buffer.readonly (*C member*), 128
Py_buffer.shape (*C member*), 128
Py_buffer.strides (*C member*), 129
Py_buffer.suboffsets (*C member*), 129
Py_BuildValue (*C function*), 94
Py_BytesMain (*C function*), 234
Py_BytesWarningFlag (*C var*), 230
Py_CHARMASK (*C macro*), 5
Py_CLEANUP_SUPPORTED (*C macro*), 91
Py_CLEAR (*C function*), 51
Py_CompileString (*C function*), 45
Py_CompileString (*C* 함수), 46
Py_CompileStringExFlags (*C function*), 45
Py_CompileStringFlags (*C function*), 45
Py_CompileStringObject (*C function*), 45
Py_complex (*C type*), 154
Py_complex.imag (*C member*), 154
Py_complex.real (*C member*), 154
Py_CONSTANT_ELLIPSIS (*C macro*), 106
Py_CONSTANT_EMPTY_BYTES (*C macro*), 106
Py_CONSTANT_EMPTY_STR (*C macro*), 106
Py_CONSTANT_EMPTY_TUPLE (*C macro*), 106
Py_CONSTANT_FALSE (*C macro*), 106
Py_CONSTANT_NONE (*C macro*), 106
Py_CONSTANT_NOT_IMPLEMENTED (*C macro*), 106
Py_CONSTANT_ONE (*C macro*), 106
Py_CONSTANT_TRUE (*C macro*), 106
Py_CONSTANT_ZERO (*C macro*), 106
PY_CXX_CONST (*C macro*), 93
Py_DEBUG (*C macro*), 12
Py_DebugFlag (*C var*), 230

Py_DecodeLocale (*C function*), 78
Py_DECREF (*C function*), 50
Py_DecRef (*C function*), 51
Py_DECREF (*C* 함수), 7
Py_DEPRECATED (*C macro*), 5
Py_DontWriteBytecodeFlag (*C var*), 231
Py_Ellipsis (*C var*), 213
Py_EncodeLocale (*C function*), 79
Py_END_ALLOW_THREADS (*C macro*), 240, 244
Py_END_CRITICAL_SECTION (*C macro*), 257
Py_END_CRITICAL_SECTION2 (*C macro*), 258
Py_EndInterpreter (*C function*), 249
Py_EnterRecursiveCall (*C function*), 64
Py_EQ (*C macro*), 336
Py_eval_input (*C var*), 46
Py_Exit (*C function*), 82
Py_ExitStatusException (*C function*), 267
Py_False (*C var*), 152
Py_FatalError (*C function*), 82
Py_FatalError(), 238
Py_fclose (*C function*), 80
Py_FdIsInteractive (*C function*), 77
Py_file_input (*C var*), 46
Py_Finalize (*C function*), 234
Py_FinalizeEx (*C function*), 233, 234, 249
Py_FinalizeEx (*C* 함수), 82
Py_fopen (*C function*), 80
Py_FrozenFlag (*C var*), 231
Py_GE (*C macro*), 336
Py_GenericAlias (*C function*), 226
Py_GenericAliasType (*C var*), 227
Py_GetArgcArgv (*C function*), 285
Py_GetBuildInfo (*C function*), 238
Py_GetCompiler (*C function*), 238
Py_GetConstant (*C function*), 105
Py_GetConstantBorrowed (*C function*), 106
Py_GetCopyright (*C function*), 237
Py_GETENV (*C macro*), 5
Py_GetExecPrefix (*C function*), 236
Py_GetExecPrefix (*C* 함수), 12
Py_GetPath (*C function*), 237
Py_GetPath (*C* 함수), 12
Py_GetPath(), 236
Py_GetPlatform (*C function*), 237
Py_GetPrefix (*C function*), 236
Py_GetPrefix (*C* 함수), 12
Py_GetProgramFullPath (*C function*), 237
Py_GetProgramFullPath (*C* 함수), 12
Py_GetProgramName (*C function*), 236
Py_GetPythonHome (*C function*), 239
Py_GetVersion (*C function*), 237
Py_GT (*C macro*), 336
Py_hash_t (*C type*), 98
Py_HashBuffer (*C function*), 99
Py_HashPointer (*C function*), 98
Py_HashRandomizationFlag (*C var*), 231
Py_IgnoreEnvironmentFlag (*C var*), 231
Py_INCREF (*C function*), 49
Py_IncRef (*C function*), 51
Py_INCREF (*C* 함수), 7
Py_Initialize (*C function*), 233, 249
Py_Initialize (*C* 함수), 12
Py_Initialize(), 236
Py_InitializeEx (*C function*), 233
Py_InitializeFromConfig (*C function*), 233
Py_InitializeFromInitConfig (*C function*), 262
Py_InspectFlag (*C var*), 231
Py_InteractiveFlag (*C var*), 231
Py_Is (*C function*), 306
Py_IS_TYPE (*C function*), 306
Py_IsFalse (*C function*), 306
Py_IsFinalizing (*C function*), 234
Py_IsInitialized (*C function*), 234
Py_IsInitialized (*C* 함수), 12
Py_IsNone (*C function*), 306
Py_IsolatedFlag (*C var*), 231
Py_IsTrue (*C function*), 306
Py_LE (*C macro*), 336
Py_LeaveRecursiveCall (*C function*), 65
Py_LegacyWindowsFSEncodingFlag (*C var*), 232
Py_LegacyWindowsStdioFlag (*C var*), 232
Py_LIMITED_API (*C macro*), 16
Py_LT (*C macro*), 336
Py_Main (*C function*), 235
PY_MAJOR_VERSION (*C macro*), 359
Py_MARSHAL_VERSION (*C macro*), 86
Py_MAX (*C macro*), 5
Py_MEMBER_SIZE (*C macro*), 5
PY_MICRO_VERSION (*C macro*), 359
Py_MIN (*C macro*), 5
PY_MINOR_VERSION (*C macro*), 359
Py_mod_create (*C macro*), 205
Py_mod_exec (*C macro*), 206
Py_mod_gil (*C macro*), 206
Py_MOD_GIL_NOT_USED (*C macro*), 206
Py_MOD_GIL_USED (*C macro*), 206
Py_mod_multiple_interpreters (*C macro*), 206
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED (*C macro*), 206
Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED (*C macro*), 206
Py_MOD_PER_INTERPRETER_GIL_SUPPORTED (*C macro*), 206
PY_MONITORING_EVENT_BRANCH_LEFT (*C macro*), 366
PY_MONITORING_EVENT_BRANCH_RIGHT (*C macro*), 366
PY_MONITORING_EVENT_C_RAISE (*C macro*), 366
PY_MONITORING_EVENT_C_RETURN (*C macro*), 366
PY_MONITORING_EVENT_CALL (*C macro*), 366
PY_MONITORING_EVENT_EXCEPTION_HANDLED (*C macro*), 366
PY_MONITORING_EVENT_INSTRUCTION (*C macro*), 366
PY_MONITORING_EVENT_JUMP (*C macro*), 366
PY_MONITORING_EVENT_LINE (*C macro*), 366

PY_MONITORING_EVENT_PY_RESUME (*C macro*), 366
PY_MONITORING_EVENT_PY_RETURN (*C macro*), 366
PY_MONITORING_EVENT_PY_START (*C macro*), 366
PY_MONITORING_EVENT_PY_THROW (*C macro*), 366
PY_MONITORING_EVENT_PY_UNWIND (*C macro*), 366
PY_MONITORING_EVENT_PY_YIELD (*C macro*), 366
PY_MONITORING_EVENT_RAISE (*C macro*), 366
PY_MONITORING_EVENT_RERAISE (*C macro*), 366
PY_MONITORING_EVENT_STOP_ITERATION (*C macro*), 366
PY_MONITORING_IS_INSTRUMENTED_EVENT (*C function*), 367
Py_NE (*C macro*), 336
Py_NewInterpreter (*C function*), 249
Py_NewInterpreterFromConfig (*C function*), 248
Py_NewRef (*C function*), 50
Py_NO_INLINE (*C macro*), 5
Py_None (*C var*), 142
Py_NoSiteFlag (*C var*), 232
Py_NotImplemented (*C var*), 106
Py_NoUserSiteDirectory (*C var*), 232
Py_OpenCodeHookFunction (*C type*), 202
Py_OptimizeFlag (*C var*), 232
Py_PACK_FULL_VERSION (*C function*), 360
Py_PACK_VERSION (*C function*), 360
Py_PreInitialize (*C function*), 269
Py_PreInitializeFromArgs (*C function*), 269
Py_PreInitializeFromBytesArgs (*C function*), 269
Py_PRINT_RAW (*C macro*), 106
Py_PRINT_RAW (*C 매크로*), 203
Py_QuietFlag (*C var*), 233
Py_READONLY (*C macro*), 311
Py_REFCNT (*C function*), 49
Py_RELATIVE_OFFSET (*C macro*), 311
PY_RELEASE_LEVEL (*C macro*), 359
PY_RELEASE_SERIAL (*C macro*), 359
Py_ReprEnter (*C function*), 65
Py_ReprLeave (*C function*), 65
Py_RETURN_FALSE (*C macro*), 152
Py_RETURN_NONE (*C macro*), 142
Py_RETURN_NOTIMPLEMENTED (*C macro*), 106
Py_RETURN_RICHCOMPARE (*C macro*), 336
Py_RETURN_TRUE (*C macro*), 152
Py_RunMain (*C function*), 235
Py_SET_REFCNT (*C function*), 49
Py_SET_SIZE (*C function*), 307
Py_SET_TYPE (*C function*), 306
Py_SetProgramName (*C function*), 236
Py_SetPythonHome (*C function*), 239
Py_SETREF (*C macro*), 51
Py_single_input (*C var*), 46
Py_SIZE (*C function*), 307
Py_ssize_t (*C type*), 10
PY_SSIZE_T_MAX (*C 매크로*), 145
Py_STRINGIFY (*C macro*), 5
Py_T_BOOL (*C macro*), 313
Py_T_BYTE (*C macro*), 313
Py_T_CHAR (*C macro*), 313
Py_T_DOUBLE (*C macro*), 313
Py_T_FLOAT (*C macro*), 313
Py_T_INT (*C macro*), 313
Py_T_LONG (*C macro*), 313
Py_T_LONGLONG (*C macro*), 313
Py_T_OBJECT_EX (*C macro*), 313
Py_T_PYSSIZET (*C macro*), 313
Py_T_SHORT (*C macro*), 313
Py_T_STRING (*C macro*), 313
Py_T_STRING_INPLACE (*C macro*), 313
Py_T_UBYTE (*C macro*), 313
Py_T_UINT (*C macro*), 313
Py_T_ULONG (*C macro*), 313
Py_T_ULONGLONG (*C macro*), 313
Py_T_USHORT (*C macro*), 313
Py_tp_token (*C macro*), 142
Py_TP_USE_SPEC (*C macro*), 142
Py_TPFLAGS_BASE_EXC_SUBCLASS (*C macro*), 330
Py_TPFLAGS_BASETYPE (*C macro*), 328
Py_TPFLAGS_BYTES_SUBCLASS (*C macro*), 330
Py_TPFLAGS_DEFAULT (*C macro*), 329
Py_TPFLAGS_DICT_SUBCLASS (*C macro*), 330
Py_TPFLAGS_DISALLOW_INSTANTIATION (*C macro*), 330
Py_TPFLAGS_HAVE_FINALIZE (*C macro*), 330
Py_TPFLAGS_HAVE_GC (*C macro*), 328
Py_TPFLAGS_HAVE_VECTORCALL (*C macro*), 330
Py_TPFLAGS_HEAPTYPE (*C macro*), 328
Py_TPFLAGS_IMMUTABLETYPE (*C macro*), 330
Py_TPFLAGS_ITEMS_AT_END (*C macro*), 329
Py_TPFLAGS_LIST_SUBCLASS (*C macro*), 330
Py_TPFLAGS_LONG_SUBCLASS (*C macro*), 330
Py_TPFLAGS_MANAGED_DICT (*C macro*), 329
Py_TPFLAGS_MANAGED_WEAKREF (*C macro*), 329
Py_TPFLAGS_MAPPING (*C macro*), 331
Py_TPFLAGS_METHOD_DESCRIPTOR (*C macro*), 329
Py_TPFLAGS_READY (*C macro*), 328
Py_TPFLAGS_READYING (*C macro*), 328
Py_TPFLAGS_SEQUENCE (*C macro*), 331
Py_TPFLAGS_TUPLE_SUBCLASS (*C macro*), 330
Py_TPFLAGS_TYPE_SUBCLASS (*C macro*), 330
Py_TPFLAGS_UNICODE_SUBCLASS (*C macro*), 330
Py_TPFLAGS_VALID_VERSION_TAG (*C macro*), 332
Py_tracefunc (*C type*), 251
Py_True (*C var*), 152
Py_tss_NEEDS_INIT (*C macro*), 254
Py_tss_t (*C type*), 254
Py_TYPE (*C function*), 306
Py_UCS1 (*C type*), 159
Py_UCS2 (*C type*), 159
Py_UCS4 (*C type*), 159
Py_uhash_t (*C type*), 98
Py_UNBLOCK_THREADS (*C macro*), 244
Py_UnbufferedStdioFlag (*C var*), 233
Py_UNICODE (*C type*), 180
Py_UNICODE_IS_HIGH_SURROGATE (*C function*), 162
Py_UNICODE_IS_LOW_SURROGATE (*C function*), 162

Py_UNICODE_IS_SURROGATE (*C function*), 162
Py_UNICODE_ISALNUM (*C function*), 162
Py_UNICODE_ISALPHA (*C function*), 162
Py_UNICODE_ISDECIMAL (*C function*), 161
Py_UNICODE_ISDIGIT (*C function*), 161
Py_UNICODE_ISLINEBREAK (*C function*), 161
Py_UNICODE_ISLOWER (*C function*), 161
Py_UNICODE_ISNUMERIC (*C function*), 162
Py_UNICODE_ISPRINTABLE (*C function*), 162
Py_UNICODE_ISSPACE (*C function*), 161
Py_UNICODE_ISTITLE (*C function*), 161
Py_UNICODE_ISUPPER (*C function*), 161
Py_UNICODE_JOIN_SURROGATES (*C function*), 162
Py_UNICODE_REPLACEMENT_CHARACTER (*C macro*), 171
Py_UNICODE_TODECIMAL (*C function*), 162
Py_UNICODE_TODIGIT (*C function*), 162
Py_UNICODE_TOLOWER (*C function*), 162
Py_UNICODE_TONUMERIC (*C function*), 162
Py_UNICODE_TOTITLE (*C function*), 162
Py_UNICODE_TOUPPER (*C function*), 162
Py_UNREACHABLE (*C macro*), 5
Py_UNUSED (*C macro*), 6
Py_VaBuildValue (*C function*), 96
PY_VECTORCALL_ARGUMENTS_OFFSET (*C macro*), 116
Py_VerboseFlag (*C var*), 233
Py_Version (*C var*), 359
PY_VERSION_HEX (*C macro*), 359
Py_VISIT (*C macro*), 357
Py_XDECREF (*C function*), 50
Py_XDECREF (*C 함수*), 11
Py_XINCREF (*C function*), 50
Py_XNewRef (*C function*), 50
Py_XSETREF (*C macro*), 51
PyAIter_Check (*C function*), 126
PyAnySet_Check (*C function*), 191
PyAnySet_CheckExact (*C function*), 191
PyArg_Parse (*C function*), 93
PyArg_ParseTuple (*C function*), 92
PyArg_ParseTupleAndKeywords (*C function*), 92
PyArg_UnpackTuple (*C function*), 93
PyArg_ValidateKeywordArguments (*C function*), 92
PyArg_VaParse (*C function*), 92
PyArg_VaParseTupleAndKeywords (*C function*), 92
PyASCIIObject (*C type*), 160
PyAsyncMethods (*C type*), 350
PyAsyncMethods.am_aiter (*C member*), 350
PyAsyncMethods.am_anext (*C member*), 350
PyAsyncMethods.am_await (*C member*), 350
PyAsyncMethods.am_send (*C member*), 350
PyBaseObject_Type (*C var*), 306
PyBool_Check (*C function*), 152
PyBool_FromLong (*C function*), 152
PyBool_Type (*C var*), 152
PyBUF_ANY_CONTIGUOUS (*C macro*), 130
PyBUF_C_CONTIGUOUS (*C macro*), 130
PyBUF_CONTIG (*C macro*), 131
PyBUF_CONTIG_RO (*C macro*), 131
PyBUF_F_CONTIGUOUS (*C macro*), 130
PyBUF_FORMAT (*C macro*), 130
PyBUF_FULL (*C macro*), 131
PyBUF_FULL_RO (*C macro*), 131
PyBUF_INDIRECT (*C macro*), 130
PyBUF_MAX_NDIM (*C macro*), 129
PyBUF_ND (*C macro*), 130
PyBUF_READ (*C macro*), 213
PyBUF_RECORDS (*C macro*), 131
PyBUF_RECORDS_RO (*C macro*), 131
PyBUF_SIMPLE (*C macro*), 130
PyBUF_STRIDED (*C macro*), 131
PyBUF_STRIDED_RO (*C macro*), 131
PyBUF_STRIDES (*C macro*), 130
PyBUF_WRITABLE (*C macro*), 129
PyBUF_WRITE (*C macro*), 213
PyBuffer_FillContiguousStrides (*C function*), 133
PyBuffer_FillInfo (*C function*), 133
PyBuffer_FromContiguous (*C function*), 133
PyBuffer_GetPointer (*C function*), 133
PyBuffer_IsContiguous (*C function*), 133
PyBuffer_Release (*C function*), 132
PyBuffer_SizeFromFormat (*C function*), 133
PyBuffer_ToContiguous (*C function*), 133
PyBufferProcs (*C type*), 349
PyBufferProcs (*C 형*), 127
PyBufferProcs.bf_getbuffer (*C member*), 349
PyBufferProcs.bf_releasebuffer (*C member*), 349
PyByteArray_AS_STRING (*C function*), 159
PyByteArray_AsString (*C function*), 159
PyByteArray_Check (*C function*), 158
PyByteArray_CheckExact (*C function*), 158
PyByteArray_Concat (*C function*), 158
PyByteArray_FromObject (*C function*), 158
PyByteArray_FromStringAndSize (*C function*), 158
PyByteArray_GET_SIZE (*C function*), 159
PyByteArray_Resize (*C function*), 159
PyByteArray_Size (*C function*), 159
PyByteArray_Type (*C var*), 158
PyByteArrayObject (*C type*), 158
PyBytes_AS_STRING (*C function*), 157
PyBytes_AsString (*C function*), 157
PyBytes_AsStringAndSize (*C function*), 157
PyBytes_Check (*C function*), 156
PyBytes_CheckExact (*C function*), 156
PyBytes_Concat (*C function*), 158
PyBytes_ConcatAndDel (*C function*), 158
PyBytes_FromFormat (*C function*), 156
PyBytes_FromFormatV (*C function*), 157
PyBytes_FromObject (*C function*), 157
PyBytes_FromString (*C function*), 156
PyBytes_FromStringAndSize (*C function*), 156
PyBytes_GET_SIZE (*C function*), 157

PyBytes_Join (C function), 158
PyBytes_Size (C function), 157
PyBytes_Type (C var), 156
PyBytesObject (C type), 156
PyCallable_Check (C function), 119
PyCallIter_Check (C function), 211
PyCallIter_New (C function), 211
PyCallIter_Type (C var), 211
PyCapsule (C type), 216
PyCapsule_CheckExact (C function), 216
PyCapsule_Destructor (C type), 216
PyCapsule_GetContext (C function), 216
PyCapsule_GetDestructor (C function), 216
PyCapsule_GetName (C function), 216
PyCapsule_GetPointer (C function), 216
PyCapsule_Import (C function), 216
PyCapsule_IsValid (C function), 217
PyCapsule_New (C function), 216
PyCapsule_SetContext (C function), 217
PyCapsule_SetDestructor (C function), 217
PyCapsule_SetName (C function), 217
PyCapsule_SetPointer (C function), 217
PyCell_Check (C function), 195
PyCell_GET (C function), 196
PyCell_Get (C function), 196
PyCell_New (C function), 196
PyCell_SET (C function), 196
PyCell_Set (C function), 196
PyCell_Type (C var), 195
PyCellObject (C type), 195
PyCF_ALLOW_TOP_LEVEL_AWAIT (C macro), 47
PyCF_ONLY_AST (C macro), 47
PyCF_OPTIMIZED_AST (C macro), 47
PyCF_TYPE_COMMENTS (C macro), 47
PyCFunction (C type), 307
PyCFunction_New (C function), 310
PyCFunction_NewEx (C function), 310
PyCFunctionFast (C type), 307
PyCFunctionFastWithKeywords (C type), 308
PyCFunctionWithKeywords (C type), 307
PyCMethod (C type), 308
PyCMethod_New (C function), 310
PyCode_Addr2Line (C function), 197
PyCode_Addr2Location (C function), 197
PyCode_AddWatcher (C function), 198
PyCode_Check (C function), 196
PyCode_ClearWatcher (C function), 198
PyCode_GetCellvars (C function), 198
PyCode_GetCode (C function), 197
PyCode_GetFreevars (C function), 198
PyCode_GetNumFree (C function), 196
PyCode_GetVarnames (C function), 198
PyCode_New (C 함수), 197
PyCode_NewEmpty (C function), 197
PyCode_NewWithPosOnlyArgs (C 함수), 197
PyCode_Type (C var), 196
PyCode_WatchCallback (C type), 198
PyCodec_BackslashReplaceErrors (C function), 102
PyCodec_Decode (C function), 100
PyCodec_Decoder (C function), 101
PyCodec_Encode (C function), 100
PyCodec_Encoder (C function), 101
PyCodec_IgnoreErrors (C function), 101
PyCodec_IncrementalDecoder (C function), 101
PyCodec_IncrementalEncoder (C function), 101
PyCodec_KnownEncoding (C function), 100
PyCodec_LookupError (C function), 101
PyCodec_NameReplaceErrors (C function), 102
PyCodec_Register (C function), 100
PyCodec_RegisterError (C function), 101
PyCodec_ReplaceErrors (C function), 101
PyCodec_StreamReader (C function), 101
PyCodec_StreamWriter (C function), 101
PyCodec_StrictErrors (C function), 101
PyCodec_Unregister (C function), 100
PyCodec_XMLCharRefReplaceErrors (C function), 102
PyCodeEvent (C type), 198
PyCodeObject (C type), 196
PyCompactUnicodeObject (C type), 160
PyCompilerFlags (C struct), 46
PyCompilerFlags.cf_feature_version (C member), 46
PyCompilerFlags.cf_flags (C member), 46
PyComplex_AsCComplex (C function), 156
PyComplex_Check (C function), 155
PyComplex_CheckExact (C function), 155
PyComplex_FromCComplex (C function), 155
PyComplex_FromDoubles (C function), 155
PyComplex_ImagAsDouble (C function), 155
PyComplex_RealAsDouble (C function), 155
PyComplex_Type (C var), 155
PyComplexObject (C type), 155
PyConfig (C type), 270
PyConfig_Clear (C function), 271
PyConfig_Get (C function), 264
PyConfig_GetInt (C function), 264
PyConfig_InitIsolatedConfig (C function), 270
PyConfig_InitPythonConfig (C function), 270
PyConfig_Names (C function), 264
PyConfig._pystats (C member), 281
PyConfig_Read (C function), 271
PyConfig_Set (C function), 264
PyConfig_SetArgv (C function), 270
PyConfig_SetBytesArgv (C function), 270
PyConfig_SetBytesString (C function), 270
PyConfig_SetString (C function), 270
PyConfig_SetWideStringList (C function), 270
PyConfig.argv (C member), 271
PyConfig.base_exec_prefix (C member), 272
PyConfig.base_executable (C member), 272
PyConfig.base_prefix (C member), 272
PyConfig.buffered_stdio (C member), 272
PyConfig.bytes_warning (C member), 272

PyConfig.check_hash_pycs_mode (C member), 273
PyConfig.code_debug_ranges (C member), 273
PyConfig.configure_c_stdio (C member), 273
PyConfig.cpu_count (C member), 276
PyConfig.dev_mode (C member), 273
PyConfig.dump_refs (C member), 273
PyConfig.dump_refs_file (C member), 273
PyConfig.exec_prefix (C member), 274
PyConfig.executable (C member), 274
PyConfig.faulthandler (C member), 274
PyConfig.filesystem_encoding (C member), 274
PyConfig.filesystem_errors (C member), 274
PyConfig.hash_seed (C member), 275
PyConfig.home (C member), 275
PyConfig.import_time (C member), 275
PyConfig.inspect (C member), 275
PyConfig.install_signal_handlers (C member), 275
PyConfig.int_max_str_digits (C member), 275
PyConfig.interactive (C member), 275
PyConfig.isolated (C member), 276
PyConfig.legacy_windows_stdio (C member), 276
PyConfig.malloc_stats (C member), 276
PyConfig.module_search_paths (C member), 277
PyConfig.module_search_paths_set (C member), 277
PyConfig.optimization_level (C member), 277
PyConfig.orig_argv (C member), 277
PyConfig.parse_argv (C member), 277
PyConfig.parser_debug (C member), 277
PyConfig.pathconfig_warnings (C member), 278
PyConfig.perf_profiling (C member), 280
PyConfig.platlibdir (C member), 276
PyConfig.prefix (C member), 278
PyConfig.program_name (C member), 278
PyConfig.pycache_prefix (C member), 278
PyConfig.pythonpath_env (C member), 277
PyConfig.quiet (C member), 278
PyConfig.run_command (C member), 278
PyConfig.run_filename (C member), 278
PyConfig.run_module (C member), 279
PyConfig.run_presite (C member), 279
PyConfig.safe_path (C member), 271
PyConfig.show_ref_count (C member), 279
PyConfig.site_import (C member), 279
PyConfig.skip_source_first_line (C member), 279
PyConfig.stdio_encoding (C member), 279
PyConfig.stdio_errors (C member), 279
PyConfig.stdlib_dir (C member), 280
PyConfig.tracemalloc (C member), 280
PyConfig.use_environment (C member), 280
PyConfig.use_frozen_modules (C member), 274
PyConfig.use_hash_seed (C member), 275
PyConfig.use_system_logger (C member), 280
PyConfig.user_site_directory (C member), 280
PyConfig.verbose (C member), 281
PyConfig.warn_default_encoding (C member), 272
PyConfig.warnoptions (C member), 281
PyConfig.write_bytecode (C member), 281
PyConfig.xoptions (C member), 281
PyContext (C type), 221
PyContext_AddWatcher (C function), 222
PyContext_CheckExact (C function), 221
PyContext_ClearWatcher (C function), 222
PyContext_Copy (C function), 221
PyContext_CopyCurrent (C function), 221
PyContext_Enter (C function), 222
PyContext_Exit (C function), 222
PyContext_New (C function), 221
PyContext_Type (C var), 221
PyContext_WatchCallback (C type), 222
PyContextEvent (C type), 222
PyContextToken (C type), 221
PyContextToken_CheckExact (C function), 221
PyContextToken_Type (C var), 221
PyContextVar (C type), 221
PyContextVar_CheckExact (C function), 221
PyContextVar_Get (C function), 222
PyContextVar_New (C function), 222
PyContextVar_Reset (C function), 223
PyContextVar_Set (C function), 223
PyContextVar_Type (C var), 221
PyCoro_CheckExact (C function), 220
PyCoro_New (C function), 220
PyCoro_Type (C var), 220
PyCoroObject (C type), 220
PyDate_Check (C function), 223
PyDate_CheckExact (C function), 223
PyDate_FromDate (C function), 224
PyDate_FromTimestamp (C function), 226
PyDateTime_Check (C function), 223
PyDateTime_CheckExact (C function), 224
PyDateTime_Date (C type), 223
PyDateTime_DATE_GET_FOLD (C function), 225
PyDateTime_DATE_GET_HOUR (C function), 225
PyDateTime_DATE_GET_MICROSECOND (C function), 225
PyDateTime_DATE_GET_MINUTE (C function), 225
PyDateTime_DATE_GET_SECOND (C function), 225
PyDateTime_DATE_GET_TZINFO (C function), 225
PyDateTime_DateTime (C type), 223
PyDateTime_DateTimeType (C var), 223
PyDateTime_DateType (C var), 223
PyDateTime_Delta (C type), 223
PyDateTime_DELTA_GET_DAYS (C function), 226
PyDateTime_DELTA_GET_MICROSECONDS (C function), 226
PyDateTime_DELTA_GET_SECONDS (C function), 226
PyDateTime_DeltaType (C var), 223
PyDateTime_FromDateAndTime (C function), 224
PyDateTime_FromDateAndTimeAndFold (C function), 224

PyDateTime_FromTimestamp (*C function*), 226
PyDateTime_GET_DAY (*C function*), 225
PyDateTime_GET_MONTH (*C function*), 225
PyDateTime_GET_YEAR (*C function*), 225
PyDateTime_Time (*C type*), 223
PyDateTime_TIME_GET_FOLD (*C function*), 225
PyDateTime_TIME_GET_HOUR (*C function*), 225
PyDateTime_TIME_GET_MICROSECOND (*C function*), 225
PyDateTime_TIME_GET_MINUTE (*C function*), 225
PyDateTime_TIME_GET_SECOND (*C function*), 225
PyDateTime_TIME_GET_TZINFO (*C function*), 226
PyDateTime_TimeType (*C var*), 223
PyDateTime_TimeZone_UTC (*C var*), 223
PyDateTime_TZInfoType (*C var*), 223
PyDelta_Check (*C function*), 224
PyDelta_CheckExact (*C function*), 224
PyDelta_FromDSU (*C function*), 224
PyDescr_IsData (*C function*), 212
PyDescr_NewClassMethod (*C function*), 212
PyDescr_NewGetSet (*C function*), 211
PyDescr_NewMember (*C function*), 211
PyDescr_NewMethod (*C function*), 211
PyDescr_NewWrapper (*C function*), 212
PyDict_AddWatcher (*C function*), 189
PyDict_Check (*C function*), 186
PyDict_CheckExact (*C function*), 186
PyDict_Clear (*C function*), 186
PyDict_ClearWatcher (*C function*), 190
PyDict_Contains (*C function*), 186
PyDict_ContainsString (*C function*), 186
PyDict_Copy (*C function*), 186
PyDict_DelItem (*C function*), 186
PyDict_DelItemString (*C function*), 186
PyDict_GetItem (*C function*), 187
PyDict_GetItemRef (*C function*), 186
PyDict_GetItemString (*C function*), 187
PyDict_GetItemStringRef (*C function*), 187
PyDict_GetItemWithError (*C function*), 187
PyDict_Items (*C function*), 188
PyDict_Keys (*C function*), 188
PyDict_Merge (*C function*), 189
PyDict_MergeFromSeq2 (*C function*), 189
PyDict_New (*C function*), 186
PyDict_Next (*C function*), 188
PyDict_Pop (*C function*), 187
PyDict_PopString (*C function*), 188
PyDict_SetDefault (*C function*), 187
PyDict_SetDefaultRef (*C function*), 187
PyDict_SetItem (*C function*), 186
PyDict_SetItemString (*C function*), 186
PyDict_Size (*C function*), 188
PyDict_Type (*C var*), 185
PyDict_Unwatch (*C function*), 190
PyDict_Update (*C function*), 189
PyDict_Values (*C function*), 188
PyDict_Watch (*C function*), 190
PyDict_WatchCallback (*C type*), 190
PyDict_WatchEvent (*C type*), 190
PyDictObject (*C type*), 185
PyDictProxy_New (*C function*), 186
PyDoc_STR (*C macro*), 6
PyDoc_STRVAR (*C macro*), 6
PyEllipsis_Type (*C var*), 213
PyErr_BadArgument (*C function*), 55
PyErr_BadInternalCall (*C function*), 56
PyErr_CheckSignals (*C function*), 61
PyErr_Clear (*C function*), 53
PyErr_Clear (*C 함수*), 10, 11
PyErr_DisplayException (*C function*), 54
PyErr_ExceptionMatches (*C function*), 58
PyErr_ExceptionMatches (*C 함수*), 11
PyErr_Fetch (*C function*), 58
PyErr_Format (*C function*), 54
PyErr_FormatUnraisable (*C function*), 54
PyErr_FormatV (*C function*), 54
PyErr_GetExcInfo (*C function*), 60
PyErr_GetHandledException (*C function*), 59
PyErr_GetRaisedException (*C function*), 58
PyErr_GivenExceptionMatches (*C function*), 58
PyErr_NewException (*C function*), 62
PyErr_NewExceptionWithDoc (*C function*), 62
PyErr_NoMemory (*C function*), 55
PyErr_NormalizeException (*C function*), 59
PyErr_Occurred (*C function*), 57
PyErr_Occurred (*C 함수*), 10
PyErr_Print (*C function*), 54
PyErr_PrintEx (*C function*), 53
PyErr_ResourceWarning (*C function*), 57
PyErr_Restore (*C function*), 59
PyErr_SetExcFromWindowsErr (*C function*), 55
PyErr_SetExcFromWindowsErrWithFilename (*C function*), 56
PyErr_SetExcFromWindowsErrWithFilenameObject (*C function*), 56
PyErr_SetExcFromWindowsErrWithFilenameObjects (*C function*), 56
PyErr_SetExcInfo (*C function*), 60
PyErr_SetFromErrno (*C function*), 55
PyErr_SetFromErrnoWithFilename (*C function*), 55
PyErr_SetFromErrnoWithFilenameObject (*C function*), 55
PyErr_SetFromErrnoWithFilenameObjects (*C function*), 55
PyErr_SetFromWindowsErr (*C function*), 55
PyErr_SetFromWindowsErrWithFilename (*C function*), 55
PyErr_SetHandledException (*C function*), 60
PyErr_SetImportError (*C function*), 56
PyErr_SetImportErrorSubclass (*C function*), 56
PyErr_SetInterrupt (*C function*), 61
PyErr_SetInterruptEx (*C function*), 61
PyErr_SetNone (*C function*), 55
PyErr_SetObject (*C function*), 54
PyErr_SetRaisedException (*C function*), 58

PyErr_SetString (*C function*), 54
PyErr_SetString (*C 함수*), 10
PyErr_SyntaxLocation (*C function*), 56
PyErr_SyntaxLocationEx (*C function*), 56
PyErr_SyntaxLocationObject (*C function*), 56
PyErr_WarnEx (*C function*), 57
PyErr_WarnExplicit (*C function*), 57
PyErr_WarnExplicitObject (*C function*), 57
PyErr_WarnFormat (*C function*), 57
PyErr_WriteUnraisable (*C function*), 54
PyEval_AcquireThread (*C function*), 247
PyEval_AcquireThread(), 242
PyEval_EvalCode (*C function*), 46
PyEval_EvalCodeEx (*C function*), 46
PyEval_EvalFrame (*C function*), 46
PyEval_EvalFrameEx (*C function*), 46
PyEval_GetBuiltins (*C function*), 99
PyEval_GetFrame (*C function*), 99
PyEval_GetFrameBuiltins (*C function*), 100
PyEval_GetFrameGlobals (*C function*), 100
PyEval_GetFrameLocals (*C function*), 100
PyEval_GetFuncDesc (*C function*), 100
PyEval_GetFuncName (*C function*), 100
PyEval_GetGlobals (*C function*), 99
PyEval_GetLocals (*C function*), 99
PyEval_InitThreads (*C function*), 242
PyEval_InitThreads(), 233
PyEval_MergeCompilerFlags (*C function*), 46
PyEval_ReleaseThread (*C function*), 247
PyEval_ReleaseThread(), 242
PyEval_RestoreThread (*C function*), 240, 242
PyEval_RestoreThread(), 242
PyEval_SaveThread (*C function*), 240, 242
PyEval_SaveThread(), 242
PyEval_SetProfile (*C function*), 252
PyEval_SetProfileAllThreads (*C function*), 252
PyEval_SetTrace (*C function*), 252
PyEval_SetTraceAllThreads (*C function*), 252
PyExc_ArithmeticError (*C var*), 65
PyExc_AssertionError (*C var*), 65
PyExc_AttributeError (*C var*), 66
PyExc_BaseException (*C var*), 65
PyExc_BaseExceptionGroup (*C var*), 65
PyExc_BlockingIOError (*C var*), 66
PyExc_BrokenPipeError (*C var*), 66
PyExc_BufferError (*C var*), 66
PyExc_BytesWarning (*C var*), 71
PyExc_ChildProcessError (*C var*), 66
PyExc_ConnectionAbortedError (*C var*), 66
PyExc_ConnectionError (*C var*), 66
PyExc_ConnectionRefusedError (*C var*), 66
PyExc_ConnectionResetError (*C var*), 66
PyExc_DeprecationWarning (*C var*), 71
PyExc_EncodingWarning (*C var*), 71
PyExc_EnvironmentError (*C var*), 70
PyExc_EOFError (*C var*), 66
PyExc_Exception (*C var*), 65
PyExc_FileExistsError (*C var*), 66
PyExc_FileNotFoundError (*C var*), 66
PyExc_FloatingPointError (*C var*), 66
PyExc_FutureWarning (*C var*), 71
PyExc_GeneratorExit (*C var*), 67
PyExc_ImportError (*C var*), 67
PyExc_ImportWarning (*C var*), 71
PyExc_IndentationError (*C var*), 67
PyExc_IndexError (*C var*), 67
PyExc_InterruptedError (*C var*), 67
PyExc_IOError (*C var*), 70
PyExc_IsADirectoryError (*C var*), 67
PyExc_KeyboardInterrupt (*C var*), 67
PyExc_KeyError (*C var*), 67
PyExc_LookupError (*C var*), 67
PyExc_MemoryError (*C var*), 67
PyExc_ModuleNotFoundError (*C var*), 67
PyExc_NameError (*C var*), 67
PyExc_NotADirectoryError (*C var*), 67
PyExc_NotImplementedError (*C var*), 68
PyExc_OSError (*C var*), 68
PyExc_OverflowError (*C var*), 68
PyExc_PendingDeprecationWarning (*C var*), 71
PyExc_PermissionError (*C var*), 68
PyExc_ProcessLookupError (*C var*), 68
PyExc_PythonFinalizationError (*C var*), 68
PyExc_RecursionError (*C var*), 68
PyExc_ReferenceError (*C var*), 68
PyExc_ResourceWarning (*C var*), 71
PyExc_RuntimeError (*C var*), 68
PyExc_RuntimeWarning (*C var*), 71
PyExc_StopAsyncIteration (*C var*), 68
PyExc_StopIteration (*C var*), 68
PyExc_SyntaxError (*C var*), 68
PyExc_SyntaxWarning (*C var*), 71
PyExc_SystemError (*C var*), 68
PyExc_SystemExit (*C var*), 68
PyExc_TabError (*C var*), 69
PyExc_TimeoutError (*C var*), 69
PyExc_TypeError (*C var*), 69
PyExc_UnboundLocalError (*C var*), 69
PyExc_UnicodeDecodeError (*C var*), 69
PyExc_UnicodeEncodeError (*C var*), 69
PyExc_UnicodeError (*C var*), 69
PyExc_UnicodeTranslateError (*C var*), 69
PyExc_UnicodeWarning (*C var*), 71
PyExc_UserWarning (*C var*), 71
PyExc_ValueError (*C var*), 69
PyExc_Warning (*C var*), 71
PyExc_WindowsError (*C var*), 70
PyExc_ZeroDivisionError (*C var*), 69
PyException_GetArgs (*C function*), 63
PyException_GetCause (*C function*), 62
PyException_GetContext (*C function*), 62
PyException_GetTraceback (*C function*), 62
PyException_SetArgs (*C function*), 63
PyException_SetCause (*C function*), 62
PyException_SetContext (*C function*), 62
PyException_SetTraceback (*C function*), 62

PyExceptionClass_Check (*C function*), 62
PyExceptionClass_Name (*C function*), 62
PyFile_FromFd (*C function*), 202
PyFile_GetLine (*C function*), 202
PyFile_SetOpenCodeHook (*C function*), 202
PyFile_WriteObject (*C function*), 202
PyFile_WriteString (*C function*), 203
PyFloat_AS_DOUBLE (*C function*), 153
PyFloat_AsDouble (*C function*), 153
PyFloat_Check (*C function*), 152
PyFloat_CheckExact (*C function*), 153
PyFloat_FromDouble (*C function*), 153
PyFloat_FromString (*C function*), 153
PyFloat_GetInfo (*C function*), 153
PyFloat_GetMax (*C function*), 153
PyFloat_GetMin (*C function*), 153
PyFloat_Pack2 (*C function*), 154
PyFloat_Pack4 (*C function*), 154
PyFloat_Pack8 (*C function*), 154
PyFloat_Type (*C var*), 152
PyFloat_Unpack2 (*C function*), 154
PyFloat_Unpack4 (*C function*), 154
PyFloat_Unpack8 (*C function*), 154
PyFloatObject (*C type*), 152
PyFrame_Check (*C function*), 218
PyFrame_GetBack (*C function*), 218
PyFrame_GetBuiltins (*C function*), 218
PyFrame_GetCode (*C function*), 218
PyFrame_GetGenerator (*C function*), 218
PyFrame_GetGlobals (*C function*), 218
PyFrame_GetLasti (*C function*), 218
PyFrame_GetLineNumber (*C function*), 219
PyFrame_GetLocals (*C function*), 219
PyFrame_GetVar (*C function*), 218
PyFrame_GetVarString (*C function*), 218
PyFrame_Type (*C var*), 218
PyFrameLocalsProxy_Check (*C function*), 219
PyFrameLocalsProxy_Type (*C var*), 219
PyFrameObject (*C type*), 217
PyFrozenSet_Check (*C function*), 191
PyFrozenSet_CheckExact (*C function*), 191
PyFrozenSet_New (*C function*), 191
PyFrozenSet_Type (*C var*), 191
PyFunction_AddWatcher (*C function*), 193
PyFunction_Check (*C function*), 192
PyFunction_ClearWatcher (*C function*), 194
PyFunction_GET_ANNOTATIONS (*C function*), 193
PyFunction_GET_CLOSURE (*C function*), 193
PyFunction_GET_CODE (*C function*), 193
PyFunction_GET_DEFAULTS (*C function*), 193
PyFunction_GET_GLOBALS (*C function*), 193
PyFunction_GET_KW_DEFAULTS (*C function*), 193
PyFunction_GET_MODULE (*C function*), 193
PyFunction_GetAnnotations (*C function*), 193
PyFunction_GetClosure (*C function*), 193
PyFunction_GetCode (*C function*), 193
PyFunction_GetDefaults (*C function*), 193
PyFunction_GetGlobals (*C function*), 193
PyFunction_GetKwDefaults (*C function*), 193
PyFunction_GetModule (*C function*), 193
PyFunction_New (*C function*), 192
PyFunction_NewWithQualName (*C function*), 192
PyFunction_SetAnnotations (*C function*), 193
PyFunction_SetClosure (*C function*), 193
PyFunction_SetDefaults (*C function*), 193
PyFunction_SetVectorcall (*C function*), 193
PyFunction_Type (*C var*), 192
PyFunction_WatchCallback (*C type*), 194
PyFunction_WatchEvent (*C type*), 194
PyFunctionObject (*C type*), 192
PyGC_Collect (*C function*), 358
PyGC_Disable (*C function*), 358
PyGC_Enable (*C function*), 358
PyGC_IsEnabled (*C function*), 358
PyGen_Check (*C function*), 220
PyGen_CheckExact (*C function*), 220
PyGen_New (*C function*), 220
PyGen_NewWithQualName (*C function*), 220
PyGen_Type (*C var*), 220
PyGenObject (*C type*), 220
PyGetSetDef (*C type*), 314
PyGetSetDef.closure (*C member*), 314
PyGetSetDef.doc (*C member*), 314
PyGetSetDef.get (*C member*), 314
PyGetSetDef.name (*C member*), 314
PyGetSetDef.set (*C member*), 314
PyGILState_Check (*C function*), 244
PyGILState_Ensure (*C function*), 243
PyGILState_GetThisThreadState (*C function*), 244
PyGILState_Release (*C function*), 244
PyHASH_BITS (*C macro*), 98
PyHash_FuncDef (*C type*), 98
PyHash_FuncDef.hash (*C member*), 98
PyHash_FuncDef.hash_bits (*C member*), 98
PyHash_FuncDef.name (*C member*), 98
PyHash_FuncDef.seed_bits (*C member*), 98
PyHash_GetFuncDef (*C function*), 98
PyHASH_IMAG (*C macro*), 98
PyHASH_INF (*C macro*), 98
PyHASH_MODULUS (*C macro*), 98
PyHASH_MULTIPLIER (*C macro*), 98
PyImport_AddModule (*C function*), 83
PyImport_AddModuleObject (*C function*), 83
PyImport_AddModuleRef (*C function*), 83
PyImport_AppendInittab (*C function*), 85
PyImport_ExecCodeModule (*C function*), 84
PyImport_ExecCodeModuleEx (*C function*), 84
PyImport_ExecCodeModuleObject (*C function*), 84
PyImport_ExecCodeModuleWithPathnames (*C function*), 84
PyImport_ExtendInittab (*C function*), 86
PyImport_FrozenModules (*C var*), 85
PyImport_GetImporter (*C function*), 85
PyImport_GetMagicNumber (*C function*), 84
PyImport_GetMagicTag (*C function*), 85

PyImport_GetModule (*C function*), 85
PyImport_GetModuleDict (*C function*), 85
PyImport_Import (*C function*), 83
PyImport_ImportFrozenModule (*C function*), 85
PyImport_ImportFrozenModuleObject (*C function*), 85
PyImport_ImportModule (*C function*), 82
PyImport_ImportModuleAttr (*C function*), 86
PyImport_ImportModuleAttrString (*C function*), 86
PyImport_ImportModuleEx (*C function*), 83
PyImport_ImportModuleLevel (*C function*), 83
PyImport_ImportModuleLevelObject (*C function*), 83
PyImport_ImportModuleNoBlock (*C function*), 82
PyImport_ReloadModule (*C function*), 83
PyIndex_Check (*C function*), 122
PyInit_modulename (*C function*), 74
PyInitConfig (*C struct*), 260
PyInitConfig_AddModule (*C function*), 262
PyInitConfig_Create (*C function*), 260
PyInitConfig_Free (*C function*), 260
PyInitConfig_FreeStrList (*C function*), 261
PyInitConfig_GetError (*C function*), 260
PyInitConfig_GetExitCode (*C function*), 260
PyInitConfig_GetInt (*C function*), 261
PyInitConfig_GetStr (*C function*), 261
PyInitConfig_GetStrList (*C function*), 261
PyInitConfig_HasOption (*C function*), 261
PyInitConfig_SetInt (*C function*), 261
PyInitConfig_SetStr (*C function*), 261
PyInitConfig_SetStrList (*C function*), 261
PyInstanceMethod_Check (*C function*), 195
PyInstanceMethod_Function (*C function*), 195
PyInstanceMethod_GET_FUNCTION (*C function*), 195
PyInstanceMethod_New (*C function*), 195
PyInstanceMethod_Type (*C var*), 194
PyInterpreterConfig (*C type*), 247
PyInterpreterConfig_DEFAULT_GIL (*C macro*), 248
PyInterpreterConfig_OWN_GIL (*C macro*), 248
PyInterpreterConfig_SHARED_GIL (*C macro*), 248
PyInterpreterConfig.allow_daemon_threads (*C member*), 248
PyInterpreterConfig.allow_exec (*C member*), 248
PyInterpreterConfig.allow_fork (*C member*), 248
PyInterpreterConfig.allow_threads (*C member*), 248
PyInterpreterConfig.check_multi_interp_extensions (*C member*), 248
PyInterpreterConfig.gil (*C member*), 248
PyInterpreterConfig.use_main_obmalloc (*C member*), 248
PyInterpreterState (*C type*), 242
PyInterpreterState_Clear (*C function*), 245
PyInterpreterState_Delete (*C function*), 245
PyInterpreterState_Get (*C function*), 246
PyInterpreterState_GetDict (*C function*), 246
PyInterpreterState_GetID (*C function*), 246
PyInterpreterState_Head (*C function*), 253
PyInterpreterState_Main (*C function*), 253
PyInterpreterState_New (*C function*), 245
PyInterpreterState_Next (*C function*), 253
PyInterpreterState_ThreadHead (*C function*), 253
PyIter_Check (*C function*), 126
PyIter_Next (*C function*), 126
PyIter_NextItem (*C function*), 126
PyIter_Send (*C function*), 127
PyList_Append (*C function*), 185
PyList_AsTuple (*C function*), 185
PyList_Check (*C function*), 183
PyList_CheckExact (*C function*), 184
PyList_Clear (*C function*), 185
PyList_Extend (*C function*), 185
PyList_GET_ITEM (*C function*), 184
PyList_GET_SIZE (*C function*), 184
PyList_GetItem (*C function*), 184
PyList_GetItem (*C 함수*), 9
PyList_GetItemRef (*C function*), 184
PyList_GetSlice (*C function*), 185
PyList_Insert (*C function*), 185
PyList_New (*C function*), 184
PyList_Reverse (*C function*), 185
PyList_SET_ITEM (*C function*), 184
PyList_SetItem (*C function*), 184
PyList_SetItem (*C 함수*), 7
PyList_SetSlice (*C function*), 185
PyList_Size (*C function*), 184
PyList_Sort (*C function*), 185
PyList_Type (*C var*), 183
PyListObject (*C type*), 183
PyLong_AS_LONG (*C function*), 144
PyLong_AsDouble (*C function*), 146
PyLong_AsInt (*C function*), 144
PyLong_AsInt32 (*C function*), 146
PyLong_AsInt64 (*C function*), 146
PyLong_AsLong (*C function*), 144
PyLong_AsLongAndOverflow (*C function*), 144
PyLong_AsLongLong (*C function*), 145
PyLong_AsLongLongAndOverflow (*C function*), 145
PyLong_AsNativeBytes (*C function*), 147
PyLong_AsSize_t (*C function*), 145
PyLong_AsSsize_t (*C function*), 145
PyLong_AsUInt32 (*C function*), 146
PyLong_AsUInt64 (*C function*), 146
PyLong_AsUnsignedLong (*C function*), 145
PyLong_AsUnsignedLongLong (*C function*), 145
PyLong_AsUnsignedLongLongMask (*C function*), 146
PyLong_AsUnsignedLongMask (*C function*), 145
PyLong_AsVoidPtr (*C function*), 146

PyLong_Check (*C function*), 142
PyLong_CheckExact (*C function*), 143
PyLong_Export (*C function*), 151
PyLong_FreeExport (*C function*), 151
PyLong_FromDouble (*C function*), 143
PyLong_FromInt32 (*C function*), 143
PyLong_FromInt64 (*C function*), 143
PyLong_FromLong (*C function*), 143
PyLong_FromLongLong (*C function*), 143
PyLong_FromNativeBytes (*C function*), 144
PyLong_FromSize_t (*C function*), 143
PyLong_FromSsize_t (*C function*), 143
PyLong_FromString (*C function*), 143
PyLong_FromUInt32 (*C function*), 143
PyLong_FromUInt64 (*C function*), 143
PyLong_FromUnicodeObject (*C function*), 144
PyLong_FromUnsignedLong (*C function*), 143
PyLong_FromUnsignedLongLong (*C function*), 143
PyLong_FromUnsignedNativeBytes (*C function*), 144
PyLong_FromVoidPtr (*C function*), 144
PyLong_GetInfo (*C function*), 149
PyLong_GetNativeLayout (*C function*), 150
PyLong_GetSign (*C function*), 149
PyLong_IsNegative (*C function*), 149
PyLong_IsPositive (*C function*), 149
PyLong_IsZero (*C function*), 149
PyLong_Type (*C var*), 142
PyLongExport (*C struct*), 151
PyLongExport.digits (*C member*), 151
PyLongExport.ndigits (*C member*), 151
PyLongExport.negative (*C member*), 151
PyLongExport.value (*C member*), 151
PyLongLayout (*C struct*), 150
PyLongLayout.bits_per_digit (*C member*), 150
PyLongLayout.digit_endianness (*C member*), 150
PyLongLayout.digit_size (*C member*), 150
PyLongLayout.digits_order (*C member*), 150
PyLongObject (*C type*), 142
PyLongWriter (*C struct*), 151
PyLongWriter_Create (*C function*), 151
PyLongWriter_Discard (*C function*), 152
PyLongWriter_Finish (*C function*), 152
PyMapping_Check (*C function*), 125
PyMapping_DelItem (*C function*), 125
PyMapping_DelItemString (*C function*), 125
PyMapping_GetItemString (*C function*), 125
PyMapping_GetOptionalItem (*C function*), 125
PyMapping_GetOptionalItemString (*C function*), 125
PyMapping_HasKey (*C function*), 125
PyMapping_HasKeyString (*C function*), 126
PyMapping_HasKeyStringWithError (*C function*), 125
PyMapping_HasKeyWithError (*C function*), 125
PyMapping_Items (*C function*), 126
PyMapping_Keys (*C function*), 126
PyMapping_Length (*C function*), 125
PyMapping_SetItemString (*C function*), 125
PyMapping_Size (*C function*), 125
PyMapping_Values (*C function*), 126
PyMappingMethods (*C type*), 347
PyMappingMethods.mp_ass_subscript (*C member*), 348
PyMappingMethods.mp_length (*C member*), 347
PyMappingMethods.mp_subscript (*C member*), 347
PyMarshal_ReadLastObjectFromFile (*C function*), 87
PyMarshal_ReadLongFromFile (*C function*), 87
PyMarshal_ReadObjectFromFile (*C function*), 87
PyMarshal_ReadObjectFromString (*C function*), 87
PyMarshal_ReadShortFromFile (*C function*), 87
PyMarshal_WriteLongToFile (*C function*), 86
PyMarshal_WriteObjectToFile (*C function*), 86
PyMarshal_WriteObjectToString (*C function*), 86
PyMem_Calloc (*C function*), 289
PyMem_Del (*C function*), 290
PYMEM_DOMAIN_MEM (*C macro*), 293
PYMEM_DOMAIN_OBJ (*C macro*), 293
PYMEM_DOMAIN_RAW (*C macro*), 293
PyMem_Free (*C function*), 290
PyMem_GetAllocator (*C function*), 293
PyMem_Malloc (*C function*), 289
PyMem_New (*C macro*), 290
PyMem_RawCalloc (*C function*), 289
PyMem_RawFree (*C function*), 289
PyMem_RawMalloc (*C function*), 288
PyMem_RawRealloc (*C function*), 289
PyMem_Realloc (*C function*), 290
PyMem_Resize (*C macro*), 290
PyMem_SetAllocator (*C function*), 293
PyMem_SetupDebugHooks (*C function*), 294
PyMemAllocatorDomain (*C type*), 293
PyMemAllocatorEx (*C type*), 292
PyMember_GetOne (*C function*), 311
PyMember_SetOne (*C function*), 311
PyMemberDef (*C type*), 310
PyMemberDef.doc (*C member*), 310
PyMemberDef.flags (*C member*), 310
PyMemberDef.name (*C member*), 310
PyMemberDef.offset (*C member*), 310
PyMemberDef.type (*C member*), 310
PyMemoryView_Check (*C function*), 214
PyMemoryView_FromBuffer (*C function*), 214
PyMemoryView_FromMemory (*C function*), 213
PyMemoryView_FromObject (*C function*), 213
PyMemoryView_GET_BASE (*C function*), 214
PyMemoryView_GET_BUFFER (*C function*), 214
PyMemoryView_GetContiguous (*C function*), 214
PyMethod_Check (*C function*), 195
PyMethod_Function (*C function*), 195
PyMethod_GET_FUNCTION (*C function*), 195
PyMethod_GET_SELF (*C function*), 195

PyMethod_New (C function), 195
PyMethod_Self (C function), 195
PyMethod_Type (C var), 195
PyMethodDef (C type), 308
PyMethodDef.ml_doc (C member), 308
PyMethodDef.ml_flags (C member), 308
PyMethodDef.ml_meth (C member), 308
PyMethodDef.ml_name (C member), 308
PyMODINIT_FUNC (C macro), 74
PyModule_Add (C function), 208
PyModule_AddFunctions (C function), 210
PyModule_AddIntConstant (C function), 209
PyModule_AddIntMacro (C macro), 209
PyModule_AddObject (C function), 209
PyModule_AddObjectRef (C function), 208
PyModule_AddStringConstant (C function), 209
PyModule_AddStringMacro (C macro), 209
PyModule_AddType (C function), 209
PyModule_Check (C function), 203
PyModule_CheckExact (C function), 203
PyModule_Create (C function), 207
PyModule_Create2 (C function), 207
PyModule_ExecDef (C function), 207
PyModule_FromDefAndSpec (C function), 207
PyModule_FromDefAndSpec2 (C function), 207
PyModule_GetDef (C function), 203
PyModule_GetDict (C function), 203
PyModule_GetFilename (C function), 204
PyModule_GetFilenameObject (C function), 203
PyModule_GetName (C function), 203
PyModule_GetNameObject (C function), 203
PyModule_GetState (C function), 203
PyModule_New (C function), 203
PyModule_NewObject (C function), 203
PyModule_SetDocString (C function), 210
PyModule_Type (C var), 203
PyModuleDef (C type), 204
PyModuleDef_Init (C function), 75
PyModuleDef_Slot (C type), 205
PyModuleDef_Slot.slot (C member), 205
PyModuleDef_Slot.value (C member), 205
PyModuleDef.m_base (C member), 204
PyModuleDef.m_clear (C member), 205
PyModuleDef.m_doc (C member), 204
PyModuleDef.m_free (C member), 205
PyModuleDef.m_methods (C member), 204
PyModuleDef.m_name (C member), 204
PyModuleDef.m_size (C member), 204
PyModuleDef.m_slots (C member), 204
PyModuleDef.m_slots.m_reload (C member), 205
PyModuleDef.m_traverse (C member), 205
PyMonitoring_EnterScope (C function), 364
PyMonitoring_ExitScope (C function), 366
PyMonitoring_FireBranchLeftEvent (C function), 364
PyMonitoring_FireBranchRightEvent (C function), 364
PyMonitoring_FireCallEvent (C function), 363
PyMonitoring_FireCRaiseEvent (C function), 364
PyMonitoring_FireCReturnEvent (C function), 364
PyMonitoring_FireExceptionHandledEvent (C function), 364
PyMonitoring_FireJumpEvent (C function), 363
PyMonitoring_FireLineEvent (C function), 363
PyMonitoring_FirePyResumeEvent (C function), 363
PyMonitoring_FirePyReturnEvent (C function), 363
PyMonitoring_FirePyStartEvent (C function), 363
PyMonitoring_FirePyThrowEvent (C function), 364
PyMonitoring_FirePyUnwindEvent (C function), 364
PyMonitoring_FirePyYieldEvent (C function), 363
PyMonitoring_FireRaiseEvent (C function), 364
PyMonitoring_FireReraiseEvent (C function), 364
PyMonitoring_FireStopIterationEvent (C function), 364
PyMonitoringState (C type), 363
PyMutex (C type), 256
PyMutex_IsLocked (C function), 256
PyMutex_Lock (C function), 256
PyMutex_Unlock (C function), 256
PyNumber_Absolute (C function), 120
PyNumber_Add (C function), 120
PyNumber_And (C function), 121
PyNumber_AsSsize_t (C function), 122
PyNumber_Check (C function), 120
PyNumber_Divmod (C function), 120
PyNumber_Float (C function), 122
PyNumber_FloorDivide (C function), 120
PyNumber_Index (C function), 122
PyNumber_InPlaceAdd (C function), 121
PyNumber_InPlaceAnd (C function), 122
PyNumber_InPlaceFloorDivide (C function), 121
PyNumber_InPlaceLshift (C function), 122
PyNumber_InPlaceMatrixMultiply (C function), 121
PyNumber_InPlaceMultiply (C function), 121
PyNumber_InPlaceOr (C function), 122
PyNumber_InPlacePower (C function), 121
PyNumber_InPlaceRemainder (C function), 121
PyNumber_InPlaceRshift (C function), 122
PyNumber_InPlaceSubtract (C function), 121
PyNumber_InPlaceTrueDivide (C function), 121
PyNumber_InPlaceXor (C function), 122
PyNumber_Invert (C function), 120
PyNumber_Long (C function), 122
PyNumber_Lshift (C function), 121
PyNumber_MatrixMultiply (C function), 120
PyNumber_Multiply (C function), 120
PyNumber_Negative (C function), 120

PyNumber_Or (*C function*), 121
PyNumber_Positive (*C function*), 120
PyNumber_Power (*C function*), 120
PyNumber_Remainder (*C function*), 120
PyNumber_Rshift (*C function*), 121
PyNumber_Subtract (*C function*), 120
PyNumber_ToBase (*C function*), 122
PyNumber_TrueDivide (*C function*), 120
PyNumber_Xor (*C function*), 121
PyNumberMethods (*C type*), 345
PyNumberMethods.nb_absolute (*C member*), 346
PyNumberMethods.nb_add (*C member*), 346
PyNumberMethods.nb_and (*C member*), 347
PyNumberMethods.nb_bool (*C member*), 346
PyNumberMethods.nb_divmod (*C member*), 346
PyNumberMethods.nb_float (*C member*), 347
PyNumberMethods.nb_floor_divide (*C member*), 347
PyNumberMethods.nb_index (*C member*), 347
PyNumberMethods.nb_inplace_add (*C member*), 347
PyNumberMethods.nb_inplace_and (*C member*), 347
PyNumberMethods.nb_inplace_floor_divide (*C member*), 347
PyNumberMethods.nb_inplace_lshift (*C member*), 347
PyNumberMethods.nb_inplace_matrix_multiply (*C member*), 347
PyNumberMethods.nb_inplace_multiply (*C member*), 347
PyNumberMethods.nb_inplace_or (*C member*), 347
PyNumberMethods.nb_inplace_power (*C member*), 347
PyNumberMethods.nb_inplace_remainder (*C member*), 347
PyNumberMethods.nb_inplace_rshift (*C member*), 347
PyNumberMethods.nb_inplace_subtract (*C member*), 347
PyNumberMethods.nb_inplace_true_divide (*C member*), 347
PyNumberMethods.nb_inplace_xor (*C member*), 347
PyNumberMethods.nb_int (*C member*), 347
PyNumberMethods.nb_invert (*C member*), 347
PyNumberMethods.nb_lshift (*C member*), 347
PyNumberMethods.nb_matrix_multiply (*C member*), 347
PyNumberMethods.nb_multiply (*C member*), 346
PyNumberMethods.nb_negative (*C member*), 346
PyNumberMethods.nb_or (*C member*), 347
PyNumberMethods.nb_positive (*C member*), 346
PyNumberMethods.nb_power (*C member*), 346
PyNumberMethods.nb_remainder (*C member*), 346
PyNumberMethods.nb_reserved (*C member*), 347
PyNumberMethods.nb_rshift (*C member*), 347
PyNumberMethods.nb_subtract (*C member*), 346
PyNumberMethods.nb_true_divide (*C member*), 347
PyNumberMethods.nb_xor (*C member*), 347
PyObject (*C type*), 305
PyObject_ASCII (*C function*), 109
PyObject_AsFileDescriptor (*C function*), 202
PyObject_Bytes (*C function*), 110
PyObject_Call (*C function*), 117
PyObject_CallFinalizer (*C function*), 305
PyObject_CallFinalizerFromDealloc (*C function*), 305
PyObject_CallFunction (*C function*), 118
PyObject_CallFunctionObjArgs (*C function*), 118
PyObject_CallMethod (*C function*), 118
PyObject_CallMethodNoArgs (*C function*), 118
PyObject_CallMethodObjArgs (*C function*), 118
PyObject_CallMethodOneArg (*C function*), 119
PyObject_CallNoArgs (*C function*), 118
PyObject_CallObject (*C function*), 118
PyObject_Calloc (*C function*), 291
PyObject_CallOneArg (*C function*), 118
PyObject_CheckBuffer (*C function*), 132
PyObject_ClearManagedDict (*C function*), 112
PyObject_ClearWeakRefs (*C function*), 215
PyObject_CopyData (*C function*), 133
PyObject_Del (*C function*), 301
PyObject_DelAttr (*C function*), 108
PyObject_DelAttrString (*C function*), 108
PyObject_DelItem (*C function*), 111
PyObject_DelItemString (*C function*), 111
PyObject_Dir (*C function*), 111
PyObject_Format (*C function*), 109
PyObject_Free (*C function*), 291
PyObject_GC_Del (*C function*), 356
PyObject_GC_IsFinalized (*C function*), 356
PyObject_GC_IsTracked (*C function*), 356
PyObject_GC_New (*C macro*), 355
PyObject_GC_NewVar (*C macro*), 355
PyObject_GC_Resize (*C macro*), 356
PyObject_GC_Track (*C function*), 356
PyObject_GC_UnTrack (*C function*), 357
PyObject_GenericGetAttr (*C function*), 108
PyObject_GenericGetDict (*C function*), 108
PyObject_GenericHash (*C function*), 99
PyObject_GenericSetAttr (*C function*), 108
PyObject_GenericSetDict (*C function*), 109
PyObject_GetAIter (*C function*), 111
PyObject_GetArenaAllocator (*C function*), 296
PyObject_GetAttr (*C function*), 107
PyObject_GetAttrString (*C function*), 107
PyObject_GetBuffer (*C function*), 132
PyObject_GetItem (*C function*), 111
PyObject_GetItemData (*C function*), 112
PyObject_GetIter (*C function*), 111
PyObject_GetOptionalAttr (*C function*), 107
PyObject_GetOptionalAttrString (*C function*), 108

PyObject_GetTypeData (*C function*), 111
PyObject_HasAttr (*C function*), 107
PyObject_HasAttrString (*C function*), 107
PyObject_HasAttrStringWithError (*C function*), 107
PyObject_HasAttrWithError (*C function*), 107
PyObject_Hash (*C function*), 110
PyObject_HashNotImplemented (*C function*), 110
PyObject_HEAD (*C macro*), 306
PyObject_HEAD_INIT (*C macro*), 307
PyObject_Init (*C function*), 299
PyObject_InitVar (*C function*), 299
PyObject_IS_GC (*C function*), 356
PyObject_IsInstance (*C function*), 110
PyObject_IsSubclass (*C function*), 110
PyObject_IsTrue (*C function*), 110
PyObject_Length (*C function*), 111
PyObject_LengthHint (*C function*), 111
PyObject_Malloc (*C function*), 291
PyObject_New (*C macro*), 299
PyObject_NewVar (*C macro*), 300
PyObject_Not (*C function*), 110
PyObject_Print (*C function*), 106
PyObject_Realloc (*C function*), 291
PyObject_Repr (*C function*), 109
PyObject_RichCompare (*C function*), 109
PyObject_RichCompareBool (*C function*), 109
PyObject_SelfIter (*C function*), 111
PyObject_SetArenaAllocator (*C function*), 296
PyObject_SetAttr (*C function*), 108
PyObject_SetAttrString (*C function*), 108
PyObject_SetItem (*C function*), 111
PyObject_Size (*C function*), 111
PyObject_Str (*C function*), 109
PyObject_Type (*C function*), 110
PyObject_TypeCheck (*C function*), 111
PyObject_VAR_HEAD (*C macro*), 306
PyObject_Vectorcall (*C function*), 119
PyObject_VectorcallDict (*C function*), 119
PyObject_VectorcallMethod (*C function*), 119
PyObject_VisitManagedDict (*C function*), 112
PyObjectArenaAllocator (*C type*), 295
PyObject.ob_refcnt (*C member*), 305
PyObject.ob_type (*C member*), 305
PyOS_AfterFork (*C function*), 78
PyOS_AfterFork_Child (*C function*), 78
PyOS_AfterFork_Parent (*C function*), 77
PyOS_BeforeFork (*C function*), 77
PyOS_CheckStack (*C function*), 78
PyOS_double_to_string (*C function*), 97
PyOS_FSPath (*C function*), 77
PyOS_getsig (*C function*), 78
PyOS_InputHook (*C var*), 44
PyOS_ReadlineFunctionPointer (*C var*), 44
PyOS_setsig (*C function*), 78
PyOS_sighandler_t (*C type*), 78
PyOS_snprintf (*C function*), 96
PyOS_stricmp (*C function*), 97
PyOS_string_to_double (*C function*), 97
PyOS_strnicmp (*C function*), 97
PyOS_strtol (*C function*), 96
PyOS_strtoul (*C function*), 96
PyOS_vsnprintf (*C function*), 96
PyPreConfig (*C type*), 267
PyPreConfig_InitIsolatedConfig (*C function*), 267
PyPreConfig_InitPythonConfig (*C function*), 267
PyPreConfig.allocator (*C member*), 267
PyPreConfig.coerce_c_locale (*C member*), 268
PyPreConfig.coerce_c_locale_warn (*C member*), 268
PyPreConfig.configure_locale (*C member*), 268
PyPreConfig.dev_mode (*C member*), 268
PyPreConfig.isolated (*C member*), 268
PyPreConfig.legacy_windows_fs_encoding (*C member*), 268
PyPreConfig.parse_argv (*C member*), 269
PyPreConfig.use_environment (*C member*), 269
PyPreConfig.utf8_mode (*C member*), 269
PyProperty_Type (*C var*), 211
PyRefTracer (*C type*), 253
PyRefTracer_CREATE (*C var*), 253
PyRefTracer_DESTROY (*C var*), 253
PyRefTracer_GetTracer (*C function*), 253
PyRefTracer_SetTracer (*C function*), 253
PyRun_AnyFile (*C function*), 43
PyRun_AnyFileEx (*C function*), 43
PyRun_AnyFileExFlags (*C function*), 43
PyRun_AnyFileFlags (*C function*), 43
PyRun_File (*C function*), 45
PyRun_FileEx (*C function*), 45
PyRun_FileExFlags (*C function*), 45
PyRun_FileFlags (*C function*), 45
PyRun_InteractiveLoop (*C function*), 44
PyRun_InteractiveLoopFlags (*C function*), 44
PyRun_InteractiveOne (*C function*), 44
PyRun_InteractiveOneFlags (*C function*), 44
PyRun_SimpleFile (*C function*), 44
PyRun_SimpleFileEx (*C function*), 44
PyRun_SimpleFileExFlags (*C function*), 44
PyRun_SimpleString (*C function*), 43
PyRun_SimpleStringFlags (*C function*), 43
PyRun_String (*C function*), 45
PyRun_StringFlags (*C function*), 45
PySendResult (*C type*), 127
PySeqIter_Check (*C function*), 211
PySeqIter_New (*C function*), 211
PySeqIter_Type (*C var*), 211
PySequence_Check (*C function*), 123
PySequence_Concat (*C function*), 123
PySequence_Contains (*C function*), 124
PySequence_Count (*C function*), 124
PySequence_DelItem (*C function*), 123
PySequence_DelSlice (*C function*), 123
PySequence_Fast (*C function*), 124
PySequence_Fast_GET_ITEM (*C function*), 124

PySequence_Fast_GET_SIZE (*C function*), 124
PySequence_Fast_ITEMS (*C function*), 124
PySequence_GetItem (*C function*), 123
PySequence_GetItem (*C 함수*), 9
PySequence_GetSlice (*C function*), 123
PySequence_In (*C function*), 124
PySequence_Index (*C function*), 124
PySequence_InPlaceConcat (*C function*), 123
PySequence_InPlaceRepeat (*C function*), 123
PySequence_ITEM (*C function*), 124
PySequence_Length (*C function*), 123
PySequence_List (*C function*), 124
PySequence_Repeat (*C function*), 123
PySequence_SetItem (*C function*), 123
PySequence_SetSlice (*C function*), 123
PySequence_Size (*C function*), 123
PySequence_Tuple (*C function*), 124
PySequenceMethods (*C type*), 348
PySequenceMethods.sq_ass_item (*C member*), 348
PySequenceMethods.sq_concat (*C member*), 348
PySequenceMethods.sq_contains (*C member*), 348
PySequenceMethods.sq_inplace_concat (*C member*), 348
PySequenceMethods.sq_inplace_repeat (*C member*), 348
PySequenceMethods.sq_item (*C member*), 348
PySequenceMethods.sq_length (*C member*), 348
PySequenceMethods.sq_repeat (*C member*), 348
PySet_Add (*C function*), 192
PySet_Check (*C function*), 191
PySet_CheckExact (*C function*), 191
PySet_Clear (*C function*), 192
PySet_Contains (*C function*), 191
PySet_Discard (*C function*), 192
PySet_GET_SIZE (*C function*), 191
PySet_New (*C function*), 191
PySet_Pop (*C function*), 192
PySet_Size (*C function*), 191
PySet_Type (*C var*), 191
PySetObject (*C type*), 190
PySignal_SetWakeupFd (*C function*), 61
PySlice_AdjustIndices (*C function*), 213
PySlice_Check (*C function*), 212
PySlice_GetIndices (*C function*), 212
PySlice_GetIndicesEx (*C function*), 212
PySlice_New (*C function*), 212
PySlice_Type (*C var*), 212
PySlice_Unpack (*C function*), 213
PyState_AddModule (*C function*), 210
PyState_FindModule (*C function*), 210
PyState_RemoveModule (*C function*), 211
PyStatus (*C type*), 266
PyStatus_Error (*C function*), 266
PyStatus_Exception (*C function*), 266
PyStatus_Exit (*C function*), 266
PyStatus_IsError (*C function*), 267
PyStatus_IsExit (*C function*), 267
PyStatus_NoMemory (*C function*), 266
PyStatus_Ok (*C function*), 266
PyStatus.err_msg (*C member*), 266
PyStatus.exitcode (*C member*), 266
PyStatus.func (*C member*), 266
PyStructSequence_Desc (*C type*), 182
PyStructSequence_Desc.doc (*C member*), 182
PyStructSequence_Desc.fields (*C member*), 182
PyStructSequence_Desc.n_in_sequence (*C member*), 182
PyStructSequence_Desc.name (*C member*), 182
PyStructSequence_Field (*C type*), 183
PyStructSequence_Field.doc (*C member*), 183
PyStructSequence_Field.name (*C member*), 183
PyStructSequence_GET_ITEM (*C function*), 183
PyStructSequence_GetItem (*C function*), 183
PyStructSequence_InitType (*C function*), 182
PyStructSequence_InitType2 (*C function*), 182
PyStructSequence_New (*C function*), 183
PyStructSequence_NewType (*C function*), 182
PyStructSequence_SET_ITEM (*C function*), 183
PyStructSequence_SetItem (*C function*), 183
PyStructSequence_UnnamedField (*C var*), 183
PySys_AddAuditHook (*C function*), 81
PySys_Audit (*C function*), 81
PySys_AuditTuple (*C function*), 81
PySys_FormatStderr (*C function*), 81
PySys_FormatStdout (*C function*), 81
PySys_GetObject (*C function*), 80
PySys_GetXOptions (*C function*), 81
PySys_ResetWarnOptions (*C function*), 80
PySys_SetArgv (*C function*), 238
PySys_SetArgvEx (*C function*), 238
PySys_SetObject (*C function*), 80
PySys_WriteStderr (*C function*), 81
PySys_WriteStdout (*C function*), 80
Python 3000 (파이썬 *3000*), **382**
Python 향상 제안
    PEP 1, 382
    PEP 7, 3, 6
    PEP 238, 375
    PEP 278, 385
    PEP 302, 379
    PEP 343, 372
    PEP 353, 10
    PEP 362, 370, 381
    PEP 383, 168, 169
    PEP 387, 15, 16
    PEP 393, 159
    PEP 411, 382
    PEP 420, 380, 382
    PEP 442, 304, 344
    PEP 443, 376
    PEP 446, 80
    PEP 451, 206
    PEP 456, 98
    PEP 483, 376

PEP 484, 369, 375, 376, 385, 386
PEP 489, 73, 74, 248
PEP 492, 370, 371, 373
PEP 498, 374
PEP 519, 382
PEP 523, 219, 246
PEP 525, 370
PEP 526, 369, 386
PEP 528, 232, 276
PEP 529, 169, 232
PEP 538, 284
PEP 539, 254
PEP 540, 284
PEP 552, 273
PEP 554, 250
PEP 578, 82
PEP 585, 376
PEP 587, 265
PEP 590, 115
PEP 623, 159
PEP 0626#out-of-process-debuggers-and-profilers, 197
PEP 634, 331, 332
PEP 649, 369
PEP 667, 99, 219
PEP 0683, 50, 377
PEP 684, 242, 250
PEP 703, 375, 376
PEP 741, 259
PEP 3116, 385
PEP 3119, 110
PEP 3121, 204
PEP 3147, 85
PEP 3151, 69
PEP 3155, 382
PYTHON_ABI_VERSION (*C macro*), 207
PYTHON_API_VERSION (*C macro*), 207
PYTHON_CPU_COUNT, 276
PYTHON_FROZEN_MODULES, 275
PYTHON_GIL, 376
PYTHON_PERF_JIT_SUPPORT, 280
PYTHON_PRESITE, 279
PYTHONCOERCECLOCALE, 284
PYTHONDEBUG, 231, 278
PYTHONDEVMODE, 273
PYTHONDONTWRITEBYTECODE, 231, 281
PYTHONDUMPREFS, 273
PYTHONDUMPREFSFILE, 273
PYTHONEXECUTABLE, 278
PYTHONFAULTHANDLER, 274
PYTHONHASHSEED, 231, 275
PYTHONHOME, 12, 231, 239, 275
Pythonic (파이썬다운), **382**
PYTHONINSPECT, 231, 275
PYTHONINTMAXSTRDIGITS, 276
PYTHONIOENCODING, 279
PYTHONLEGACYWINDOWSFSENCODING, 232, 269
PYTHONLEGACYWINDOWSSTDIO, 232, 276
PYTHONMALLOC, 288, 292, 294, 295
PYTHONMALLOCSTATS, 276, 288
PYTHONNODEBUGRANGES, 273
PYTHONNOUSERSITE, 232, 281
PYTHONOPTIMIZE, 232, 277
PYTHONPATH, 12, 231, 277
PYTHONPERFSUPPORT, 280
PYTHONPLATLIBDIR, 276
PYTHONPROFILEIMPORTTIME, 275
PYTHONPYCACHEPREFIX, 278
PYTHONSAFEPATH, 272
PYTHONTRACEMALLOC, 280
PYTHONUNBUFFERED, 233, 272
PYTHONUTF8, 269, 284
PYTHONVERBOSE, 233, 281
PYTHONWARNINGS, 281
PyThread_create_key (*C function*), 255
PyThread_delete_key (*C function*), 255
PyThread_delete_key_value (*C function*), 255
PyThread_get_key_value (*C function*), 255
PyThread_ReInitTLS (*C function*), 255
PyThread_set_key_value (*C function*), 255
PyThread_tss_alloc (*C function*), 254
PyThread_tss_create (*C function*), 255
PyThread_tss_delete (*C function*), 255
PyThread_tss_free (*C function*), 254
PyThread_tss_get (*C function*), 255
PyThread_tss_is_created (*C function*), 255
PyThread_tss_set (*C function*), 255
PyThreadState (*C type*), 239, 242
PyThreadState_Clear (*C function*), 245
PyThreadState_Delete (*C function*), 245
PyThreadState_DeleteCurrent (*C function*), 245
PyThreadState_EnterTracing (*C function*), 246
PyThreadState_Get (*C function*), 243
PyThreadState_GetDict (*C function*), 246
PyThreadState_GetFrame (*C function*), 245
PyThreadState_GetID (*C function*), 245
PyThreadState_GetInterpreter (*C function*), 245
PyThreadState_GetUnchecked (*C function*), 243
PyThreadState_LeaveTracing (*C function*), 246
PyThreadState_New (*C function*), 245
PyThreadState_Next (*C function*), 253
PyThreadState_SetAsyncExc (*C function*), 247
PyThreadState_Swap (*C function*), 243
PyThreadState.interp (*C member*), 242
PyTime_AsSecondsDouble (*C function*), 103
PyTime_Check (*C function*), 224
PyTime_CheckExact (*C function*), 224
PyTime_FromTime (*C function*), 224
PyTime_FromTimeAndFold (*C function*), 224
PyTime_MAX (*C var*), 102
PyTime_MIN (*C var*), 102
PyTime_Monotonic (*C function*), 102
PyTime_MonotonicRaw (*C function*), 103
PyTime_PerfCounter (*C function*), 102
PyTime_PerfCounterRaw (*C function*), 103
PyTime_t (*C type*), 102

PyTime_Time (*C function*), 102
PyTime_TimeRaw (*C function*), 103
PyTimeZone_FromOffset (*C function*), 224
PyTimeZone_FromOffsetAndName (*C function*), 225
PyTrace_C_CALL (*C var*), 252
PyTrace_C_EXCEPTION (*C var*), 252
PyTrace_C_RETURN (*C var*), 252
PyTrace_CALL (*C var*), 251
PyTrace_EXCEPTION (*C var*), 251
PyTrace_LINE (*C var*), 252
PyTrace_OPCODE (*C var*), 252
PyTrace_RETURN (*C var*), 252
PyTraceMalloc_Track (*C function*), 296
PyTraceMalloc_Untrack (*C function*), 296
PyTuple_Check (*C function*), 181
PyTuple_CheckExact (*C function*), 181
PyTuple_GET_ITEM (*C function*), 181
PyTuple_GET_SIZE (*C function*), 181
PyTuple_GetItem (*C function*), 181
PyTuple_GetSlice (*C function*), 181
PyTuple_New (*C function*), 181
PyTuple_Pack (*C function*), 181
PyTuple_SET_ITEM (*C function*), 181
PyTuple_SetItem (*C function*), 181
PyTuple_SetItem (*C 함수*), 7
PyTuple_Size (*C function*), 181
PyTuple_Type (*C var*), 181
PyTupleObject (*C type*), 181
PyType_AddWatcher (*C function*), 136
PyType_Check (*C function*), 135
PyType_CheckExact (*C function*), 135
PyType_ClearCache (*C function*), 135
PyType_ClearWatcher (*C function*), 136
PyType_Freeze (*C function*), 140
PyType_FromMetaclass (*C function*), 139
PyType_FromModuleAndSpec (*C function*), 139
PyType_FromSpec (*C function*), 140
PyType_FromSpecWithBases (*C function*), 140
PyType_GenericAlloc (*C function*), 137
PyType_GenericNew (*C function*), 137
PyType_GetBaseByToken (*C function*), 138
PyType_GetDict (*C function*), 136
PyType_GetFlags (*C function*), 135
PyType_GetFullyQualifiedName (*C function*), 137
PyType_GetModule (*C function*), 138
PyType_GetModuleByDef (*C function*), 138
PyType_GetModuleName (*C function*), 137
PyType_GetModuleState (*C function*), 138
PyType_GetName (*C function*), 137
PyType_GetQualName (*C function*), 137
PyType_GetSlot (*C function*), 138
PyType_GetTypeDataSize (*C function*), 112
PyType_HasFeature (*C function*), 136
PyType_IS_GC (*C function*), 136
PyType_IsSubtype (*C function*), 137
PyType_Modified (*C function*), 136
PyType_Ready (*C function*), 137
PyType_Slot (*C type*), 141
PyType_Slot.pfunc (*C member*), 141
PyType_Slot.slot (*C member*), 141
PyType_Spec (*C type*), 140
PyType_Spec.basicsize (*C member*), 140
PyType_Spec.flags (*C member*), 141
PyType_Spec.itemsize (*C member*), 140
PyType_Spec.name (*C member*), 140
PyType_Spec.slots (*C member*), 141
PyType_Type (*C var*), 135
PyType_Watch (*C function*), 136
PyType_WatchCallback (*C type*), 136
PyTypeObject (*C type*), 135
PyTypeObject.tp_alloc (*C member*), 340
PyTypeObject.tp_as_async (*C member*), 325
PyTypeObject.tp_as_buffer (*C member*), 327
PyTypeObject.tp_as_mapping (*C member*), 326
PyTypeObject.tp_as_number (*C member*), 326
PyTypeObject.tp_as_sequence (*C member*), 326
PyTypeObject.tp_base (*C member*), 337
PyTypeObject.tp_bases (*C member*), 341
PyTypeObject.tp_basicsize (*C member*), 322
PyTypeObject.tp_cache (*C member*), 341
PyTypeObject.tp_call (*C member*), 326
PyTypeObject.tp_clear (*C member*), 333
PyTypeObject.tp_dealloc (*C member*), 322
PyTypeObject.tp_del (*C member*), 342
PyTypeObject.tp_descr_get (*C member*), 338
PyTypeObject.tp_descr_set (*C member*), 338
PyTypeObject.tp_dict (*C member*), 338
PyTypeObject.tp_dictoffset (*C member*), 339
PyTypeObject.tp_doc (*C member*), 332
PyTypeObject.tp_finalize (*C member*), 342
PyTypeObject.tp_flags (*C member*), 328
PyTypeObject.tp_free (*C member*), 340
PyTypeObject.tp_getattr (*C member*), 325
PyTypeObject.tp_getattro (*C member*), 327
PyTypeObject.tp_getset (*C member*), 337
PyTypeObject.tp_hash (*C member*), 326
PyTypeObject.tp_init (*C member*), 339
PyTypeObject.tp_is_gc (*C member*), 341
PyTypeObject.tp_itemsize (*C member*), 322
PyTypeObject.tp_iter (*C member*), 337
PyTypeObject.tp_iternext (*C member*), 337
PyTypeObject.tp_members (*C member*), 337
PyTypeObject.tp_methods (*C member*), 337
PyTypeObject.tp_mro (*C member*), 341
PyTypeObject.tp_name (*C member*), 321
PyTypeObject.tp_new (*C member*), 340
PyTypeObject.tp_repr (*C member*), 325
PyTypeObject.tp_richcompare (*C member*), 335
PyTypeObject.tp_setattr (*C member*), 325
PyTypeObject.tp_setattro (*C member*), 327
PyTypeObject.tp_str (*C member*), 327
PyTypeObject.tp_subclasses (*C member*), 341
PyTypeObject.tp_traverse (*C member*), 332
PyTypeObject.tp_vectorcall (*C member*), 344
PyTypeObject.tp_vectorcall_offset (*C member*), 324

PyTypeObject.tp_version_tag (*C member*), 342
PyTypeObject.tp_watched (*C member*), 345
PyTypeObject.tp_weaklist (*C member*), 342
PyTypeObject.tp_weaklistoffset (*C member*), 336
PyTZInfo_Check (*C function*), 224
PyTZInfo_CheckExact (*C function*), 224
PyUnicode_1BYTE_DATA (*C function*), 160
PyUnicode_1BYTE_KIND (*C macro*), 160
PyUnicode_2BYTE_DATA (*C function*), 160
PyUnicode_2BYTE_KIND (*C macro*), 160
PyUnicode_4BYTE_DATA (*C function*), 160
PyUnicode_4BYTE_KIND (*C macro*), 160
PyUnicode_Append (*C function*), 166
PyUnicode_AppendAndDel (*C function*), 166
PyUnicode_AsASCIIString (*C function*), 174
PyUnicode_AsCharmapString (*C function*), 174
PyUnicode_AsEncodedString (*C function*), 171
PyUnicode_AsLatin1String (*C function*), 174
PyUnicode_AsMBCSString (*C function*), 175
PyUnicode_AsRawUnicodeEscapeString (*C function*), 174
PyUnicode_AsUCS4 (*C function*), 167
PyUnicode_AsUCS4Copy (*C function*), 168
PyUnicode_AsUnicodeEscapeString (*C function*), 173
PyUnicode_AsUTF8 (*C function*), 172
PyUnicode_AsUTF8AndSize (*C function*), 171
PyUnicode_AsUTF8String (*C function*), 171
PyUnicode_AsUTF16String (*C function*), 173
PyUnicode_AsUTF32String (*C function*), 172
PyUnicode_AsWideChar (*C function*), 170
PyUnicode_AsWideCharString (*C function*), 170
PyUnicode_BuildEncodingMap (*C function*), 166
PyUnicode_Check (*C function*), 160
PyUnicode_CHECK_INTERNED (*C function*), 178
PyUnicode_CheckExact (*C function*), 160
PyUnicode_Compare (*C function*), 177
PyUnicode_CompareWithASCIIString (*C function*), 177
PyUnicode_Concat (*C function*), 175
PyUnicode_Contains (*C function*), 178
PyUnicode_CopyCharacters (*C function*), 167
PyUnicode_Count (*C function*), 177
PyUnicode_DATA (*C function*), 160
PyUnicode_Decode (*C function*), 171
PyUnicode_DecodeASCII (*C function*), 174
PyUnicode_DecodeCharmap (*C function*), 174
PyUnicode_DecodeCodePageStateful (*C function*), 175
PyUnicode_DecodeFSDefault (*C function*), 169
PyUnicode_DecodeFSDefaultAndSize (*C function*), 169
PyUnicode_DecodeLatin1 (*C function*), 174
PyUnicode_DecodeLocale (*C function*), 168
PyUnicode_DecodeLocaleAndSize (*C function*), 168
PyUnicode_DecodeMBCS (*C function*), 175
PyUnicode_DecodeMBCSStateful (*C function*), 175
PyUnicode_DecodeRawUnicodeEscape (*C function*), 174
PyUnicode_DecodeUnicodeEscape (*C function*), 173
PyUnicode_DecodeUTF7 (*C function*), 173
PyUnicode_DecodeUTF7Stateful (*C function*), 173
PyUnicode_DecodeUTF8 (*C function*), 171
PyUnicode_DecodeUTF8Stateful (*C function*), 171
PyUnicode_DecodeUTF16 (*C function*), 173
PyUnicode_DecodeUTF16Stateful (*C function*), 173
PyUnicode_DecodeUTF32 (*C function*), 172
PyUnicode_DecodeUTF32Stateful (*C function*), 172
PyUnicode_EncodeCodePage (*C function*), 175
PyUnicode_EncodeFSDefault (*C function*), 169
PyUnicode_EncodeLocale (*C function*), 168
PyUnicode_Equal (*C function*), 177
PyUnicode_EqualToUTF8 (*C function*), 177
PyUnicode_EqualToUTF8AndSize (*C function*), 177
PyUnicode_Fill (*C function*), 167
PyUnicode_Find (*C function*), 176
PyUnicode_FindChar (*C function*), 176
PyUnicode_Format (*C function*), 178
PyUnicode_FromEncodedObject (*C function*), 166
PyUnicode_FromFormat (*C function*), 163
PyUnicode_FromFormatV (*C function*), 166
PyUnicode_FromKindAndData (*C function*), 163
PyUnicode_FromObject (*C function*), 166
PyUnicode_FromOrdinal (*C function*), 166
PyUnicode_FromString (*C function*), 163
PyUnicode_FromStringAndSize (*C function*), 163
PyUnicode_FromWideChar (*C function*), 170
PyUnicode_FSConverter (*C function*), 169
PyUnicode_FSDecoder (*C function*), 169
PyUnicode_GET_LENGTH (*C function*), 160
PyUnicode_GetDefaultEncoding (*C function*), 166
PyUnicode_GetLength (*C function*), 166
PyUnicode_InternFromString (*C function*), 178
PyUnicode_InternInPlace (*C function*), 178
PyUnicode_IS_ASCII (*C function*), 161
PyUnicode_IS_READY (*C function*), 180
PyUnicode_IsIdentifier (*C function*), 161
PyUnicode_Join (*C function*), 176
PyUnicode_KIND (*C function*), 160
PyUnicode_MAX_CHAR_VALUE (*C function*), 161
PyUnicode_New (*C function*), 162
PyUnicode_Partition (*C function*), 176
PyUnicode_READ (*C function*), 161
PyUnicode_READ_CHAR (*C function*), 161
PyUnicode_ReadChar (*C function*), 167
PyUnicode_READY (*C function*), 180
PyUnicode_Replace (*C function*), 177
PyUnicode_Resize (*C function*), 167
PyUnicode_RichCompare (*C function*), 178
PyUnicode_RPartition (*C function*), 176
PyUnicode_RSplit (*C function*), 176

PyUnicode_Split (*C function*), 175
PyUnicode_Splitlines (*C function*), 176
PyUnicode_Substring (*C function*), 167
PyUnicode_Tailmatch (*C function*), 176
PyUnicode_Translate (*C function*), 175
PyUnicode_Type (*C var*), 159
PyUnicode_WRITE (*C function*), 160
PyUnicode_WriteChar (*C function*), 167
PyUnicodeDecodeError_Create (*C function*), 63
PyUnicodeDecodeError_GetEncoding (*C function*), 63
PyUnicodeDecodeError_GetEnd (*C function*), 64
PyUnicodeDecodeError_GetObject (*C function*), 63
PyUnicodeDecodeError_GetReason (*C function*), 64
PyUnicodeDecodeError_GetStart (*C function*), 63
PyUnicodeDecodeError_SetEnd (*C function*), 64
PyUnicodeDecodeError_SetReason (*C function*), 64
PyUnicodeDecodeError_SetStart (*C function*), 63
PyUnicodeEncodeError_GetEncoding (*C function*), 63
PyUnicodeEncodeError_GetEnd (*C function*), 64
PyUnicodeEncodeError_GetObject (*C function*), 63
PyUnicodeEncodeError_GetReason (*C function*), 64
PyUnicodeEncodeError_GetStart (*C function*), 63
PyUnicodeEncodeError_SetEnd (*C function*), 64
PyUnicodeEncodeError_SetReason (*C function*), 64
PyUnicodeEncodeError_SetStart (*C function*), 63
PyUnicodeIter_Type (*C var*), 159
PyUnicodeObject (*C type*), 160
PyUnicodeTranslateError_GetEnd (*C function*), 64
PyUnicodeTranslateError_GetObject (*C function*), 63
PyUnicodeTranslateError_GetReason (*C function*), 64
PyUnicodeTranslateError_GetStart (*C function*), 63
PyUnicodeTranslateError_SetEnd (*C function*), 64
PyUnicodeTranslateError_SetReason (*C function*), 64
PyUnicodeTranslateError_SetStart (*C function*), 63
PyUnicodeWriter (*C type*), 178
PyUnicodeWriter_Create (*C function*), 179
PyUnicodeWriter_DecodeUTF8Stateful (*C function*), 180
PyUnicodeWriter_Discard (*C function*), 179
PyUnicodeWriter_Finish (*C function*), 179
PyUnicodeWriter_Format (*C function*), 180
PyUnicodeWriter_WriteASCII (*C function*), 179
PyUnicodeWriter_WriteChar (*C function*), 179
PyUnicodeWriter_WriteRepr (*C function*), 180
PyUnicodeWriter_WriteStr (*C function*), 179
PyUnicodeWriter_WriteSubstring (*C function*), 180
PyUnicodeWriter_WriteUCS4 (*C function*), 179
PyUnicodeWriter_WriteUTF8 (*C function*), 179
PyUnicodeWriter_WriteWideChar (*C function*), 179
PyUnstable, 15
PyUnstable_AtExit (*C function*), 235
PyUnstable_Code_GetExtra (*C function*), 201
PyUnstable_Code_GetFirstFree (*C function*), 196
PyUnstable_Code_New (*C function*), 196
PyUnstable_Code_NewWithPosOnlyArgs (*C function*), 197
PyUnstable_Code_SetExtra (*C function*), 201
PyUnstable_EnableTryIncRef (*C function*), 114
PyUnstable_Eval_RequestCodeExtraIndex (*C function*), 201
PyUnstable_Exc_PrepReraiseStar (*C function*), 63
PyUnstable_GC_VisitObjects (*C function*), 358
PyUnstable_InterpreterFrame_GetCode (*C function*), 219
PyUnstable_InterpreterFrame_GetLasti (*C function*), 219
PyUnstable_InterpreterFrame_GetLine (*C function*), 220
PyUnstable_IsImmortal (*C function*), 113
PyUnstable_Long_CompactValue (*C function*), 150
PyUnstable_Long_IsCompact (*C function*), 150
PyUnstable_Module_SetGIL (*C function*), 210
PyUnstable_Object_ClearWeakRefsNoCallbacks (*C function*), 215
PyUnstable_Object_EnableDeferredRefcount (*C function*), 112
PyUnstable_Object_GC_NewWithExtraData (*C function*), 355
PyUnstable_Object_IsUniquelyReferenced (*C function*), 115
PyUnstable_Object_IsUniqueReferencedTemporary (*C function*), 112
PyUnstable_PerfMapState_Fini (*C function*), 104
PyUnstable_PerfMapState_Init (*C function*), 103
PyUnstable_TryIncRef (*C function*), 113
PyUnstable_Type_AssignVersionTag (*C function*), 138
PyUnstable_WritePerfMapEntry (*C function*), 103
PyVarObject (*C type*), 305
PyVarObject_HEAD_INIT (*C macro*), 307
PyVarObject.ob_size (*C member*), 305
PyVectorcall_Call (*C function*), 117
PyVectorcall_Function (*C function*), 117
PyVectorcall_NARGS (*C function*), 117
PyWeakref_Check (*C function*), 214
PyWeakref_CheckProxy (*C function*), 214
PyWeakref_CheckRef (*C function*), 214
PyWeakref_GET_OBJECT (*C function*), 215

PyWeakref_GetObject (*C function*), 215
PyWeakref_GetRef (*C function*), 215
PyWeakref_IsDead (*C function*), 215
PyWeakref_NewProxy (*C function*), 214
PyWeakref_NewRef (*C function*), 214
PyWideStringList (*C type*), 266
PyWideStringList_Append (*C function*), 266
PyWideStringList_Insert (*C function*), 266
PyWideStringList.items (*C member*), 266
PyWideStringList.length (*C member*), 266
PyWrapper_New (*C function*), 212

Q

qualified name (정규화된 이름), **382**

R

READ_RESTRICTED (*C* 매크로), 311
READONLY (*C* 매크로), 311
realloc (*C* 함수), 287
reference count (참조 횟수), **383**
regular package (정규 패키지), **383**
releasebufferproc (*C type*), 351
REPL, **383**
repr
    built-in function, 325
    내장 함수, 109
reprfunc (*C type*), 351
RESTRICTED (*C* 매크로), 311
richcmpfunc (*C type*), 351

S

search
    path, module, 233, 237
sendfunc (*C type*), 352
sequence
    object, 156
sequence (시퀀스), **383**
set
    객체, 190
set comprehension (집합 컴프리헨션), **383**
set_all(), 8
setattrfunc (*C type*), 351
setattrofunc (*C type*), 351
setswitchinterval (*in module sys*), 239
setter (*C type*), 314
SIGINT (*C* 매크로), 61
single dispatch (싱글 디스패치), **383**
SIZE_MAX (*C* 매크로), 145
slice (슬라이스), **384**
soft deprecated (약하게 폐지된), **384**
special
    method (메서드), 384
special method (특수 메서드), **384**
ssizeargfunc (*C type*), 352
ssizeobjargproc (*C type*), 352
standard library, **384**
statement (문장), **384**
static type checker (정적 형 검사기), **384**
staticmethod
    내장 함수, 309
stderr (*in module sys*), 248, 249
stdin (*in module sys*), 248, 249
stdlib, **384**
stdout (*in module sys*), 248, 249
strerror (*C* 함수), 55
strong reference (강한 참조), **384**
structmember.h, 314
sum_list(), 9
sum_sequence(), 9, 10
sys
    module, 233, 248, 249
    모듈, 12
SystemError (내장 예외), 203, 204

T

t-string, **384**
t-strings, **384**
T_BOOL (*C* 매크로), 314
T_BYTE (*C* 매크로), 314
T_CHAR (*C* 매크로), 314
T_DOUBLE (*C* 매크로), 314
T_FLOAT (*C* 매크로), 314
T_INT (*C* 매크로), 314
T_LONG (*C* 매크로), 314
T_LONGLONG (*C* 매크로), 314
T_NONE (*C macro*), 314
T_OBJECT (*C macro*), 314
T_OBJECT_EX (*C* 매크로), 314
T_PYSSIZET (*C* 매크로), 314
T_SHORT (*C* 매크로), 314
T_STRING (*C* 매크로), 314
T_STRING_INPLACE (*C* 매크로), 314
T_UBYTE (*C* 매크로), 314
T_UINT (*C* 매크로), 314
T_ULONG (*C* 매크로), 314
T_ULONGULONG (*C* 매크로), 314
T_USHORT (*C* 매크로), 314
ternaryfunc (*C type*), 352
text encoding (텍스트 인코딩), **384**
text file (텍스트 파일), **384**
thread state, **385**
token (토큰), **385**
traverseproc (*C type*), 357
triple-quoted string (삼중 따옴표 된 문자열), **385**
type (형), **385**
type alias (형 에일리어스), **385**
type hint (형 힌트), **385**

U

ULONG_MAX (*C* 매크로), 145
unaryfunc (*C type*), 351
universal newlines (유니버설 줄 넘김), **385**
USE_STACKCHECK (*C* 매크로), 78

V

variable annotation (변수 어노테이션), **386**
vectorcallfunc (*C type*), 116
version (*in module sys*), 237, 238
virtual environment (가상 환경), **386**
virtual machine (가상 기계), **386**
visitproc (*C type*), 357

W

walrus operator, **386**
WRITE_RESTRICTED (*C* 매크로), 311

X

내장 함수
    __import__, 83
    abs, 120
    classmethod, 309
    divmod, 120
    float, 122
    int, 122
    len, 111, 123, 125, 184, 188, 191
    pow, 120, 122
    repr, 109
    staticmethod, 309
    바이트 열, 110
    아스 키, 109
    컴파일, 84
    튜플, 124, 185
    해시, 110
    형, 110
딕셔너리
    객체, 185
리스트
    객체, 183
메서드
    객체, 195
모듈
    __main__, 12
    builtins, 12
    sys, 12
    객체, 203
    검색 경로, 12
    시그널, 61
문자열
    PyObject_Str (*C* 함수), 109
바이트 배열
    객체, 158
바이트 열
    객체, 156
    내장 함수, 110
버퍼 객체
    (버퍼 프로토콜을 보십시오), 127
버퍼 인터페이스
    (버퍼 프로토콜을 보십시오), 127
버퍼 프로토콜, 127
복소수
    객체, 154
부동 소수점
    객체, 152

Y

튜플
    객체, 181
    내장 함수, 124, 185
파일
    객체, 202
패키지 변수
    __all__, 82
포트란 연속, 130
함수
    객체, 192
해시
    내장 함수, 110
형
    객체, 6, 135
    내장 함수, 110
환경 변수
    __PYVENV_LAUNCHER__, 272, 278
    PATH, 12
    PYTHON_CPU_COUNT, 276
    PYTHON_FROZEN_MODULES, 275
    PYTHON_GIL, 376
    PYTHON_PERF_JIT_SUPPORT, 280
    PYTHON_PRESITE, 279
    PYTHONCOERCECLOCALE, 284
    PYTHONDEBUG, 231, 278
    PYTHONDEVMODE, 273
    PYTHONDONTWRITEBYTECODE, 231, 281
    PYTHONDUMPREFS, 273
    PYTHONDUMPREFSFILE, 273
    PYTHONEXECUTABLE, 278
    PYTHONFAULTHANDLER, 274
    PYTHONHASHSEED, 231, 275
    PYTHONHOME, 12, 231, 239, 275
    PYTHONINSPECT, 231, 275
    PYTHONINTMAXSTRDIGITS, 276
    PYTHONIOENCODING, 279
    PYTHONLEGACYWINDOWSFSENCODING, 232, 269
    PYTHONLEGACYWINDOWSSTDIO, 232, 276
    PYTHONMALLOC, 288, 292, 294, 295
    PYTHONMALLOCSTATS, 276, 288
    PYTHONNODEBUGRANGES, 273
    PYTHONNOUSERSITE, 232, 281
    PYTHONOPTIMIZE, 232, 277
    PYTHONPATH, 12, 231, 277
    PYTHONPERFSUPPORT, 280
    PYTHONPLATLIBDIR, 276
    PYTHONPROFILEIMPORTTIME, 275
    PYTHONPYCACHEPREFIX, 278
    PYTHONSAFEPATH, 272
    PYTHONTRACEMALLOC, 280
    PYTHONUNBUFFERED, 233, 272
    PYTHONUTF8, 269, 284
    PYTHONVERBOSE, 233, 281
    PYTHONWARNINGS, 281

Z

Zen of Python (파이썬 젠), 386