

---

# Extending and Embedding Python

출시 버전 3.8.13

Guido van Rossum  
and the Python development team

3월 17, 2022



<b>1</b>	<b>권장 제삼자 도구</b>	<b>3</b>
<b>2</b>	<b>제삼자 도구 없이 확장 만들기</b>	<b>5</b>
2.1	C나 C++로 파이썬 확장하기 . . . . .	5
2.2	확장형 정의하기: 자습서 . . . . .	23
2.3	확장형 정의하기: 여러 가지 주제 . . . . .	46
2.4	C와 C++ 확장 빌드하기 . . . . .	55
2.5	윈도우에서 C와 C++ 확장 빌드하기 . . . . .	57
<b>3</b>	<b>더 큰 응용 프로그램에 CPython 런타임을 내장하기</b>	<b>61</b>
3.1	다른 응용 프로그램에 파이썬 내장하기 . . . . .	61
<b>A</b>	<b>용어집</b>	<b>67</b>
<b>B</b>	<b>이 설명서에 관하여</b>	<b>81</b>
B.1	파이썬 설명서의 공헌자들 . . . . .	81
<b>C</b>	<b>역사와 라이선스</b>	<b>83</b>
C.1	소프트웨어의 역사 . . . . .	83
C.2	파이썬에 액세스하거나 사용하기 위한 이용 약관 . . . . .	84
C.3	포함된 소프트웨어에 대한 라이선스 및 승인 . . . . .	88
<b>D</b>	<b>저작권</b>	<b>101</b>
	<b>색인</b>	<b>103</b>



이 문서는 새로운 모듈로 파이썬 인터프리터를 확장하기 위해 C 나 C++로 모듈을 작성하는 방법을 설명합니다. 이러한 모듈은 새로운 함수뿐만 아니라 새로운 객체 형과 메서드를 정의할 수 있습니다. 또한, 확장 언어로 사용하기 위해, 파이썬 인터프리터를 다른 응용 프로그램에 내장시키는 방법에 관해서도 설명합니다. 마지막으로, 하부 운영 체제에서 이 기능을 지원하는 경우, 동적으로 (실행시간에) 인터프리터에 로드될 수 있도록 확장 모듈을 컴파일하고 링크하는 방법을 보여줍니다.

이 문서는 파이썬에 대한 기본 지식을 전제로 합니다. 언어에 대한 형식적이지 않은 소개는 [tutorial-index](#) 를 보십시오. [reference-index](#) 는 보다 형식적인 언어 정의를 제공합니다. [library-index](#) 는 존재하는 객체 형, 함수 및 모듈(내장된 것과 파이썬으로 작성된 것 모두)을 설명하는데, 이것들이 언어의 응용 범위를 넓힙니다.

전체 파이썬/C API에 대한 자세한 설명은 별도의 [c-api-index](#) 를 참조하십시오.



# CHAPTER 1

---

## 권장 제삼자 도구

---

이 지침서는 이 버전의 CPython의 일부로 제공되는, 확장을 만들기 위한 기본 도구만을 다룹니다. Cython, cffi, SWIG 와 Numba 와 같은 제삼자 도구는 파이썬을 위한 C와 C++ 확장을 만드는 더 간단하고 세련된 접근법을 제공합니다.

### 더 보기:

**파이썬 패키징 사용자 지침서: 바이너리 확장** 파이썬 패키징 사용자 지침서는 바이너리 확장의 생성을 단순화하는 몇 가지 사용 가능한 도구를 다루고 있을 뿐만 아니라, 확장 모듈을 만드는 것이 왜 바람직한지 여러 가지 이유에 대해서도 논의합니다.





---

## 제삼자 도구 없이 확장 만들기

---

지침서의 이 부분에서는 제삼자 도구의 도움 없이 C와 C++ 확장을 만드는 방법에 대해 설명합니다. 여러분 자신의 C 확장을 만드는 데 권장되는 방법이라기보다는, 주로 도구를 제작하는 사람들을 대상으로 합니다.

### 2.1 C나 C++로 파이썬 확장하기

C로 프로그래밍하는 방법을 알고 있다면, 파이썬에 새로운 내장 모듈을 추가하기는 매우 쉽습니다. 그러한 확장 모듈 (*extension modules*)은 파이썬에서 직접 할 수 없는 두 가지 일을 할 수 있습니다: 새로운 내장 객체형을 구현할 수 있고, C 라이브러리 함수와 시스템 호출을 호출할 수 있습니다.

확장을 지원하기 위해, 파이썬 API(Application Programmers Interface)는 파이썬 런타임 시스템의 대부분 측면에 액세스 할 수 있는 함수, 매크로 및 변수 집합을 정의합니다. 파이썬 API는 헤더 "Python.h"를 포함해 C 소스 파일에 통합됩니다.

확장 모듈의 컴파일은 시스템 설정뿐만 아니라 의도하는 용도에 따라 다릅니다; 자세한 내용은 다음 장에서 설명합니다.

---

**참고:** C 확장 인터페이스는 CPython에만 해당하며, 확장 모듈은 다른 파이썬 구현에서는 작동하지 않습니다. 많은 경우에, C 확장을 작성하지 않고 다른 구현으로의 이식성을 유지하는 것이 가능합니다. 예를 들어, 사용 사례가 C 라이브러리 함수나 시스템 호출을 호출하는 것이라면, 사용자 정의 C 코드를 작성하는 대신 ctypes 모듈이나 cffi 라이브러리 사용을 고려해야 합니다. 이 모듈을 사용하면 C 코드와 인터페이스하기 위한 파이썬 코드를 작성할 수 있으며 C 확장 모듈을 작성하고 컴파일하는 것보다 파이썬 구현 간에 이식성이 더 좋습니다.

---

## 2.1.1 간단한 예

spam(몬티 파이썬 팬들이 가장 좋아하는 음식...)이라는 확장 모듈을 만듭시다, 그리고 C 라이브러리 함수 `system()`에 대한 파이썬 인터페이스를 만들고 싶다고 합시다<sup>1</sup>. 이 함수는 널 종료 문자열을 인자로 취하고 정수를 반환합니다. 우리는 이 함수를 다음과 같이 파이썬에서 호출할 수 있기를 원합니다:

```
>>> import spam
>>> status = spam.system("ls -l")
```

`spammodule.c` 파일을 만드는 것으로 시작하십시오. (역사적으로, 모듈을 `spam`이라고 하면, 해당 구현을 포함하는 C 파일은 `spammodule.c`라고 합니다; 모듈 이름이 `spammify`처럼 매우 길면, 모듈 이름은 그냥 `spammify.c`일 수 있습니다.)

파일의 처음 두 줄은 다음과 같습니다:

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

이것은 파이썬 API를 가져옵니다 (원한다면 모듈의 목적과 저작권 표시를 설명하는 주석을 추가할 수 있습니다).

**참고:** 파이썬이 일부 시스템의 표준 헤더에 영향을 미치는 일부 전처리기 정의를 정의할 수 있어서, 표준 헤더가 포함되기 전에 반드시 `Python.h`를 포함해야 합니다.

`Python.h`를 포함하기 전에 항상 `PY_SSIZE_T_CLEAN`을 정의하는 것이 좋습니다. 이 매크로에 대한 설명은 확장 함수에서 매개 변수 추출하기를 참조하십시오.

`Python.h`가 정의한 사용자가 볼 수 있는 기호는 표준 헤더 파일에 정의된 기호를 제외하고 모두 `Py`나 `PY` 접두사를 갖습니다. 편의를 위해, 그리고 파이썬 인터프리터가 광범위하게 사용하기 때문에, "`Python.h`"는 몇 가지 표준 헤더 파일을 포함합니다: `<stdio.h>`, `<string.h>`, `<errno.h>` 및 `<stdlib.h>`. 후자의 헤더 파일이 시스템에 없으면, 함수 `malloc()`, `free()` 및 `realloc()`을 직접 선언합니다.

다음으로 모듈 파일에 추가하는 것은 파이썬 표현식 `spam.system(string)`이 평가될 때 호출될 C 함수입니다 (이것이 어떻게 호출되는지 곧 보게 될 것입니다):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

파이썬의 인자 목록(예를 들어, 단일 표현식 "`ls -l`")에서 C 함수로 전달되는 인자로의 간단한 변환이 있습니다. C 함수에는 항상 `self`와 `args`라는 두 개의 인자가 있습니다.

`self` 인자는 모듈 수준 함수에서 모듈 객체를 가리킵니다; 메서드의 경우 객체 인스턴스를 가리킵니다.

`args` 인자는 인자를 포함하는 파이썬 튜플 객체에 대한 포인터입니다. 튜플의 각 항목은 호출의 인자 목록에 있는 인자에 해당합니다. 인자는 파이썬 객체입니다 — C 함수에서 무언가를 수행하려면 이들을 C 값으로 변환해야 합니다. 파이썬 API의 `PyArg_ParseTuple()` 함수는 인자 형을 확인하고 C 값으로 변환합니다. 템플릿 문자열을 사용하여 필요한 인자 형과 변환된 값을 저장할 C 변수 형을 결정합니다. 나중에 이것에 대해 자세히 알아보겠습니다.

모든 인자의 형이 올바르게 해당 구성 요소가 주소가 전달된 변수에 저장되면, `PyArg_ParseTuple()`은 참(0이 아닙니다)을 반환합니다. 유효하지 않은 인자 목록이 전달되면 거짓(0)을 반환합니다. 후자의 경우 호출 함수가(예에서 보듯이) `NULL`을 즉시 반환할 수 있도록 적절한 예외를 발생시킵니다.

<sup>1</sup> 이 함수에 대한 인터페이스는 표준 모듈 `os`에 이미 존재합니다 — 간단하고 단순한 예제로 선택되었습니다.

## 2.1.2 막간극: 에러와 예외

파이썬 인터프리터 전체에서 중요한 규칙은 다음과 같습니다: 함수가 실패하면 예외 조건을 설정하고 에러값(보통 NULL 포인터)을 반환해야 합니다. 예외는 인터프리터 내부의 정적 전역 변수에 저장됩니다; 이 변수가 NULL이면 예외가 발생하지 않은 것입니다. 두 번째 전역 변수는 예외의 《연관된 값》(raise에 대한 두 번째 인자)을 저장합니다. 세 번째 변수에는 에러가 파이썬 코드에서 발생한 경우 스택 트레이스백이 포함됩니다. 이 세 변수는 `sys.exc_info()`의 파이썬 결과에 대한 C 동등물입니다(파이썬 라이브러리 레퍼런스에 있는 모듈 `sys`에 대한 섹션을 참조하십시오). 에러가 어떻게 전달되는지 이해하기 위해서는 이들에 대해 아는 것이 중요합니다.

파이썬 API는 다양한 형의 예외를 설정하기 위한 여러 함수를 정의합니다.

가장 일반적인 것은 `PyErr_SetString()`입니다. 인자는 예외 객체와 C 문자열입니다. 예외 객체는 보통 `PyExc_ZeroDivisionError`와 같은 미리 정의된 객체입니다. C 문자열은 에러의 원인을 나타내며 파이썬 문자열 객체로 변환되어 예외의 《연관된 값》으로 저장됩니다.

또 다른 유용한 함수는 `PyErr_SetFromErrno()`입니다. 이 함수는 예외 인자만 취하고 전역 변수 `errno`를 검사하여 관련 값을 구성합니다. 가장 일반적인 함수는 `PyErr_SetObject()`이며, 예외와 관련 값인 두 개의 객체 인자를 취합니다. 이러한 함수들에 전달되는 객체를 `Py_INCREF()` 할 필요는 없습니다.

`PyErr_Occurred()`로 예외가 설정되어 있는지 비 파괴적으로 검사할 수 있습니다. 현재 예외 객체나 예외가 발생하지 않았으면 NULL을 반환합니다. 반환 값에서 알 수 있어야 해서 일반적으로 함수 호출에서 에러가 발생했는지 확인하기 위해 `PyErr_Occurred()`를 호출할 필요는 없습니다.

다른 함수 `g`를 호출하는 함수 `f`가 `g`의 실패를 감지할 때, `f` 자체가 에러값(보통 NULL이나 -1)을 반환해야 합니다. `PyErr_*`() 함수 중 하나를 호출하지 않아야 합니다 — `g`에 의해 이미 호출되었습니다. 그러면 `f`의 호출자도 역시 `PyErr_*`() 호출 없이, 자신의 호출자에게 에러 표시를 반환하고, 이런 식으로 계속된다고 가정합니다 — 에러를 가장 먼저 감지한 함수에 의해 에러의 가장 자세한 원인이 이미 보고되었습니다. 일단 에러가 파이썬 인터프리터의 메인 루프에 도달하면, 현재 실행 중인 파이썬 코드를 중단하고 파이썬 프로그래머가 지정한 예외 처리기를 찾으려고 시도합니다.

(모듈이 실제로 다른 `PyErr_*`() 함수를 호출하여 더 자세한 에러 메시지를 표시할 수 있는 상황이 있습니다. 그럴 때는 그렇게 하는 것이 좋습니다. 그러나, 일반적인 규칙으로 이는 필요하지 않고, 에러가 발생하는 원인에 관한 정보를 잃어버리게 합니다: 대부분의 연산은 다양한 이유로 실패할 수 있습니다.)

실패한 함수 호출로 설정된 예외를 무시하려면, `PyErr_Clear()`를 호출하여 예외 조건을 명시적으로 지워야 합니다. C 코드가 `PyErr_Clear()`를 호출해야 하는 유일한 때는 에러를 인터프리터에 전달하지 않고 스스로 완전히 처리하려고 하는 경우입니다(아마 다른 것을 시도하거나, 아무것도 잘못되지 않은 척해서).

모든 실패한 `malloc()` 호출은 예외로 전환되어야 합니다 — `malloc()`(또는 `realloc()`)의 직접 호출자는 스스로 `PyErr_NoMemory()`를 호출하고 실패 표시기를 반환해야 합니다. 모든 객체 생성 함수(예를 들어, `PyLong_FromLong()`)는 이미 이 작업을 수행하므로, 이 주의는 `malloc()`을 직접 호출하는 호출자에게만 해당합니다.

또한 `PyArg_ParseTuple()`과 그 친구들의 중요한 예외를 제외하고, 정수 상태를 반환하는 함수는 유닉스 시스템 호출처럼 일반적으로 성공 시 양수 값이나 0을 반환하고, 실패 시 -1을 반환합니다.

마지막으로, 에러 표시기를 반환할 때(이미 만든 객체를 `Py_XDECREF()`나 `Py_DECREF()`를 호출하여) 가비지를 정리하십시오!

어떤 예외를 발생시킬지는 전적으로 여러분의 것입니다. 모든 내장 파이썬 예외에 해당하는 사전 선언된 C 객체(가령 `PyExc_ZeroDivisionError`)가 있는데, 직접 사용할 수 있습니다. 물론, 예외를 현명하게 선택해야 합니다 — 파일을 열 수 없음을 뜻하는 데 `PyExc_TypeError`를 사용하지 마십시오(아마도 `PyExc_IOError`여야 합니다). 인자 목록에 문제가 있으면, `PyArg_ParseTuple()` 함수는 일반적으로 `PyExc_TypeError`를 발생시킵니다. 값이 특정 범위 내에 있어야 하거나 다른 조건을 만족해야 하는 인자가 있으면, `PyExc_ValueError`가 적합합니다.

모듈에 고유한 새 예외를 정의할 수도 있습니다. 이를 위해, 일반적으로 파일 시작 부분에 정적 객체 변수를 선언합니다:

```
static PyObject *SpamError;
```

그리고 모듈의 초기화 함수(`PyInit_spam()`)에서 예외 객체로 초기화합니다:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_XINCREF(SpamError);
    if (PyModule_AddObject(m, "error", SpamError) < 0) {
        Py_XDECREF(SpamError);
        Py_CLEAR(SpamError);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

예외 객체의 파이썬 이름은 `spam.error`임에 유의하십시오. `PyErr_NewException()` 함수는 (NULL 대신 다른 클래스가 전달되지 않는 한) 베이스 클래스가 (builtin-exceptions에서 설명된) `Exception`인 클래스를 만들 수 있습니다.

`SpamError` 변수는 새로 만들어진 예외 클래스에 대한 참조를 보유함에도 유의하십시오; 이것은 의도적입니다! 외부 코드에 의해 예외가 모듈에서 제거될 수 있기 때문에, 클래스가 버려져서 `SpamError`가 매달린 (dangling) 포인터가 되지 않도록 하려면, 클래스에 대한 참조를 소유할 필요가 있습니다. 매달린 포인터가 되면, 예외를 발생시키는 C 코드가 코어 덤프나 다른 의도하지 않은 부작용을 일으킬 수 있습니다.

이 샘플의 뒷부분에서 `PyMODINIT_FUNC`를 함수 반환형으로 사용하는 것에 관해 설명합니다.

다음과 같이 `PyErr_SetString()`을 호출하여 확장 모듈에서 `spam.error` 예외를 발생시킬 수 있습니다:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}
```

### 2.1.3 예제로 돌아가기

예제 함수로 돌아가서, 이제 여러분은 이 문장을 이해할 수 있어야 합니다:

```
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

인자 목록에서 에러가 발견되면 `PyArg_ParseTuple()`에 의해 설정된 예외에 의존하면서 `NULL`(객체 포인터를 반환하는 함수의 에러 표시기)을 반환합니다. 그렇지 않으면 인자의 문자열 값이 지역 변수 `command`에 복사되었습니다. 이것은 포인터 대입이며 가리키는 문자열을 수정해서는 안 됩니다(따라서 표준 C에서, 변수 `command`는 `const char *command`로 올바르게 선언되어야 합니다).

다음 문장은 유닉스 함수 `system()`을 호출인데, `PyArg_ParseTuple()`에서 얻은 문자열을 전달합니다:

```
sts = system(command);
```

우리의 `spam.system()` 함수는 `sts`의 값을 파이썬 객체로 반환해야 합니다. 이것은 `PyLong_FromLong()` 함수를 사용하여 이루어집니다.

```
return PyLong_FromLong(sts);
```

이 경우, 정수 객체를 반환합니다. (예, 정수조차도 파이썬에서는 힙 상의 객체입니다!)

유용한 인자를 반환하지 않는 C 함수(`void`를 반환하는 함수)가 있으면, 해당 파이썬 함수는 `None`을 반환해야 합니다. 그렇게 하려면 이 관용구가 필요합니다(`Py_RETURN_NONE` 매크로로 구현됩니다):

```
Py_INCREF(Py_None);
return Py_None;
```

`Py_None`은 특수 파이썬 객체 `None`의 C 이름입니다. 앞에서 보았듯이, 대부분의 상황에서 《에러》를 뜻하는 `NULL` 포인터가 아니라 진짜 파이썬 객체입니다.

### 2.1.4 모듈의 메서드 테이블과 초기화 함수

파이썬 프로그램에서 `spam_system()`이 어떻게 호출되는지 보여 주겠다고 약속했습니다. 먼저, 《메서드 테이블》에 이름과 주소를 나열해야 합니다:

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system", spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL} /* Sentinel */
};
```

세 번째 항목(`METH_VARARGS`)에 유의하십시오. 이것은 인터프리터에게 C 함수에 사용될 호출 규칙을 알려주는 플래그입니다. 일반적으로 항상 `METH_VARARGS`나 `METH_VARARGS | METH_KEYWORDS`여야 합니다; 0 값은 더는 사용되지 않는 `PyArg_ParseTuple()` 변형이 사용됨을 의미합니다.

`METH_VARARGS`만 사용할 때, 함수는 파이썬 수준 매개 변수가 `PyArg_ParseTuple()`을 통한 구문 분석에 허용되는 튜플로 전달될 것으로 기대해야 합니다; 이 함수에 대한 자세한 정보는 아래에 제공됩니다.

키워드 인자를 함수에 전달해야 하면, 세 번째 필드에서 `METH_KEYWORDS` 비트를 설정할 수 있습니다. 이 경우, C 함수는 키워드 딕셔너리가 될 세 번째 `PyObject *` 매개 변수를 받아들여야 합니다. 이러한 함수에는 `PyArg_ParseTupleAndKeywords()`를 사용하여 인자를 구문 분석하십시오.

메서드 테이블은 모듈 정의 구조체에서 참조되어야 합니다:

```
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam", /* name of module */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    spam_doc, /* module documentation, may be NULL */
    -1,      /* size of per-interpreter state of the module,
              or -1 if the module keeps state in global variables. */
    SpamMethods
};
    
```

다시, 이 구조체는 모듈의 초기화 함수에서 인터프리터로 전달되어야 합니다. 초기화 함수의 이름은 `PyInit_name()` 이어야 합니다, 여기서 *name*은 모듈의 이름이며, 모듈 파일에 정의된 유일한 비 static 항목이어야 합니다:

```

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModule_Create(&spammodule);
}
    
```

`PyMODINIT_FUNC`는 함수를 `PyObject *` 반환형으로 선언하고, 플랫폼에 필요한 특수 링크 선언을 선언하며, C++의 경우 함수를 `extern "C"`로 선언함에 유의하십시오.

파이썬 프로그램이 처음으로 모듈 `spam`을 импорт 할 때, `PyInit_spam()` 이 호출됩니다. (파이썬 내장에 대해서는 아래에서 언급합니다.) 이는 `PyModule_Create()` 를 호출하는데, 모듈 객체를 반환하고 모듈 정의에서 찾은 테이블(`PyMethodDef` 구조체의 배열)을 기반으로 내장 함수 객체들을 새로 만든 모듈에 삽입합니다. `PyModule_Create()` 는 만든 모듈 객체에 대한 포인터를 반환합니다. 특정 에러의 경우 치명적인 에러로 중단되거나, 모듈을 만족스럽게 초기화할 수 없으면 `NULL`을 반환할 수 있습니다. 초기화 함수는 모듈 객체를 호출자에게 반환해야 합니다. 그러면 `sys.modules`에 삽입됩니다.

파이썬을 내장할 때, `PyImport_Inittab` 테이블에 항목이 없으면 `PyInit_spam()` 함수가 자동으로 호출되지 않습니다. 모듈을 초기화 테이블에 추가하려면, `PyImport_AppendInittab()` 을 사용하고, 선택적으로 그다음에 모듈을 импорт 합니다:

```

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }

    /* Add a built-in module, before Py_Initialize */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built modules table\n");
        exit(1);
    }

    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(program);

    /* Initialize the Python interpreter.  Required.
       If this step fails, it will be a fatal error. */
    Py_Initialize();

    /* Optionally import the module; alternatively,
       import can be deferred until the embedded script
       imports it. */
    PyObject *pmodule = PyImport_ImportModule("spam");
    if (!pmodule) {
        PyErr_Print();
        fprintf(stderr, "Error: could not import module 'spam'\n");
    }
}
    
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```
...
PyMem_RawFree(program);
return 0;
}
```

**참고:** `sys.modules`에서 항목을 제거하거나 프로세스 내에서 컴파일된 모듈을 여러 인터프리터로 импорт 하면 (또는 `exec()`를 개입시키지 않고 `fork()`를 따르면) 일부 확장 모듈에 문제가 발생할 수 있습니다. 확장 모듈 작성자는 내부 데이터 구조를 초기화할 때 주의를 기울여야 합니다.

더욱 실질적인 예제 모듈이 `Modules/xxmodule.c`로 파이썬 소스 배포판에 포함되어 있습니다. 이 파일은 템플릿으로 사용되거나 단순히 예제로 읽을 수 있습니다.

**참고:** `spam` 예제와 달리 `xxmodule`은 다단계 초기화(*multi-phase initialization*)(파이썬 3.5의 새로운 기능)를 사용합니다. 여기서는 `PyModuleDef` 구조체가 `PyInit_spam`에서 반환되고, 모듈 생성은 импорт 절차에 맡겨집니다. 다단계 초기화에 대한 자세한 내용은 **PEP 489**를 참조하십시오.

## 2.1.5 컴파일과 링크

새로운 확장을 사용하기 전에 해야 할 두 가지 작업이 더 있습니다: 컴파일과 파이썬 시스템과의 링크. 동적 로딩을 사용하면, 세부 사항은 시스템이 사용하는 동적 로딩 스타일에 따라 달라질 수 있습니다; 확장 모듈을 빌드하는 것에 관한 장(**C와 C++ 확장 빌드하기** 장)과 윈도우 빌드에 대한 자세한 정보는 이에만 관련된 추가 정보(윈도우에서 **C와 C++ 확장 빌드하기** 장)를 참조하십시오.

동적 로딩을 사용할 수 없거나, 모듈을 파이썬 인터프리터의 영구적인 부분으로 만들려면, 구성 설정을 변경하고 인터프리터를 다시 빌드해야 합니다. 운 좋게도, 이것은 유닉스에서 매우 간단합니다: 압축을 풀 소스 배포의 `Modules/` 디렉터리에 파일(예를 들어 `spammodule.c`)을 넣고, `Modules/Setup.local` 파일에 여러분의 파일을 기술하는 한 줄을 추가하십시오:

```
spam spammodule.o
```

그리고 최상위 디렉터리에서 **make**를 실행하여 인터프리터를 다시 빌드하십시오. `Modules/` 서브 디렉터리에서 **make**를 실행할 수도 있지만, 먼저 **<make Makefile>**을 실행하여 `Makefile`을 다시 빌드해야 합니다. (이것은 `Setup` 파일을 변경할 때마다 필요합니다.)

모듈에 링크할 추가 라이브러리가 필요하다면, 이것도 구성 파일의 줄에 나열될 수 있습니다, 예를 들어:

```
spam spammodule.o -lX11
```

## 2.1.6 C에서 파이썬 함수 호출하기

지금까지 파이썬에서 C 함수를 호출할 수 있도록 하는 데 집중했습니다. 그 반대도 유용합니다: C에서 파이썬 함수 호출하기. 이것은 특히 **<콜백>** 함수를 지원하는 라이브러리의 경우에 해당합니다. C 인터페이스가 콜백을 사용하면, 동등한 파이썬은 종종 파이썬 프로그래머에게 콜백 메커니즘을 제공해야 할 필요가 있습니다; 구현은 C 콜백에서 파이썬 콜백 함수를 호출해야 합니다. 다른 용도도 상상할 수 있습니다.

다행히, 파이썬 인터프리터는 재귀적으로 쉽게 호출되며, 파이썬 함수를 호출하는 표준 인터페이스가 있습니다. (특정 문자열을 입력으로 파이썬 과서를 호출하는 방법에 대해서는 다루지 않겠습니다 — 관심이 있다면, 파이썬 소스 코드에서 `Modules/main.c`의 `-c` 명령 줄 옵션 구현을 살펴보십시오.)

파이썬 함수를 호출하기는 쉽습니다. 먼저, 파이썬 프로그램은 어떻게 든 여러분에게 파이썬 함수 객체를 전달해야 합니다. 이를 위해 함수(또는 다른 인터페이스)를 제공해야 합니다. 이 함수가 호출될 때, 전역 변수(또는 여러분이 보기에 적절한 곳 어디에나)에 파이썬 함수 객체에 대한 포인터를 저장하십시오 (`Py_INCREF()` 해야 하는 것에 주의하십시오!). 예를 들어, 다음 함수는 모듈 정의의 일부일 수 있습니다:

```

static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);          /* Add a reference to new callback */
        Py_XDECREF(my_callback);  /* Dispose of previous callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}

```

이 함수는 METH\_VARARGS 플래그를 사용하여 인터프리터에 등록해야 합니다; 이것은 섹션 모듈의 메서드 테이블과 초기화 함수에 설명되어 있습니다. PyArg\_ParseTuple() 함수와 그것의 인자는 확장 함수에서 매개 변수 추출하기 섹션에 설명되어 있습니다.

매크로 Py\_XINCREF()와 Py\_XDECREF()는 객체의 참조 횟수를 증가/감소시키며 NULL 포인터가 있을 때 안전합니다 (그러나 이 문맥에서 temp는 NULL이 아님에 유의하십시오). 섹션 참조 횟수에 이에 대한 자세한 정보가 있습니다.

나중에, 함수를 호출할 때, C 함수 PyObject\_CallObject()를 호출합니다. 이 함수에는 두 개의 인자가 있는데, 모두 임의의 파이썬 객체에 대한 포인터입니다: 파이썬 함수와 인자 목록. 인자 목록은 항상 길이가 인자의 수인 튜플 객체여야 합니다. 인자 없이 파이썬 함수를 호출하려면, NULL이나 빈 튜플을 전달하십시오; 하나의 인자로 호출하려면, 단 항목 튜플을 전달하십시오. Py\_BuildValue()는 포맷 문자열이 괄호 사이에 0개 이상의 포맷 코드로 구성되어 있을 때 튜플을 반환합니다. 예를 들면:

```

int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);

```

PyObject\_CallObject()는 파이썬 객체 포인터를 반환합니다: 이것은 파이썬 함수의 반환 값입니다. PyObject\_CallObject()는 인자와 관련하여 《참조 횟수 중립적》입니다. 이 예에서는 PyObject\_CallObject() 호출 직후 Py\_DECREF()되는 인자 목록으로 사용할 새 튜플이 만들어졌습니다.

PyObject\_CallObject()의 반환 값은 《새것》입니다: 완전히 새로운 객체이거나 참조 횟수가 증가한 기존 객체입니다. 따라서, 전역 변수에 저장하려는 것이 아닌 한, 설사 (특히!) 그 값에 관심이 없더라도 결과를 Py\_DECREF()해야 합니다.

그러나, 이 작업을 수행하기 전에 반환 값이 NULL이 아닌지 확인해야 합니다. 그렇다면, 파이썬 함수는 예외를 발생시켜 종료한 것입니다. PyObject\_CallObject()라는 C 코드가 파이썬에서 호출되었다면 이제 파이썬 호출자에게 여러 표시를 반환하여, 인터프리터가 스택 트레이스를 인쇄하거나 호출하는 파이썬 코드가 예외를 처리할 수 있도록 합니다. 이것이 불가능하거나 바람직하지 않으면, PyErr\_Clear()를 호출하여 예외를 지워야 합니다. 예를 들면:



```

if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);

```

파이썬 콜백 함수에 대해 원하는 인터페이스에 따라, `PyObject_CallObject()`에 인자 목록을 제공해야 할 수도 있습니다. 때에 따라 인자 목록은 콜백 함수를 지정한 같은 인터페이스를 통해 파이썬 프로그램에서 제공됩니다. 그런 다음 함수 객체와 같은 방식으로 저장하고 사용할 수 있습니다. 다른 경우에는, 인자 목록으로 전달할 새 튜플을 구성해야 할 수도 있습니다. 이렇게 하는 가장 간단한 방법은 `Py_BuildValue()`를 호출하는 것입니다. 예를 들어, 정수 이벤트 코드를 전달하려면, 다음 코드를 사용할 수 있습니다:

```

PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);

```

호출 직후, 에러 점검 전에 `Py_DECREF(arglist)`의 배치에 유의하십시오! 또한 엄격하게 말하면 이 코드가 완전하지 않음에도 유의하십시오: `Py_BuildValue()`에 메모리가 부족할 수 있어서 확인해야 합니다.

인자와 키워드 인자를 지원하는 `PyObject_Call()`을 사용하여 키워드 인자가 있는 함수를 호출할 수도 있습니다. 위의 예제와 같이, `Py_BuildValue()`를 사용하여 딕셔너리를 구성합니다.

```

PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);

```

## 2.1.7 확장 함수에서 매개 변수 추출하기

`PyArg_ParseTuple()` 함수는 다음과 같이 선언됩니다:

```

int PyArg_ParseTuple(PyObject *arg, const char *format, ...);

```

`arg` 인자는 파이썬에서 C 함수로 전달되는 인자 목록이 포함된 튜플 객체여야 합니다. `format` 인자는 포맷 문자열이어야 하며, 문법은 파이썬/C API 레퍼런스 매뉴얼의 `arg-parsing`에 설명되어 있습니다. 나머지 인자는 포맷 문자열에 의해 형이 결정되는 변수의 주소여야 합니다.

`PyArg_ParseTuple()`은 파이썬 인자가 요구되는 형인지 확인하지만, 호출에 전달된 C 변수 주소의 유효성을 확인할 수는 없습니다: 실수를 하면, 코드가 충돌하거나 적어도 메모리의 임의 비트를 덮어씁니다. 그러니 조심하십시오!

호출자에게 제공되는 모든 파이썬 객체 참조는 빌려온(*borrowed*) 참조임에 유의하십시오; 참조 횟수를 줄이지 마십시오!

몇 가지 예제 호출:

```

#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>

```

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
/* Python call: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* A string */
/* Possible Python call: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
/* Possible Python call: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
/* A pair of ints and a string, whose size is also returned */
/* Possible Python call: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* A string, and optionally another string and an integer */
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
        &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
```

## 2.1.8 확장 함수를 위한 키워드 매개 변수

PyArg\_ParseTupleAndKeywords() 함수는 다음과 같이 선언됩니다:

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
    const char *format, char *kwlist[], ...);
```

*arg*와 *format* 매개 변수는 PyArg\_ParseTuple() 함수와 동일합니다. *kwdict* 매개 변수는 파이썬 런타임에서 세 번째 매개 변수로 수신된 키워드 딕셔너리입니다. *kwlist* 매개 변수는 매개 변수를 식별하는 문자열의 NULL 종료 목록입니다; 이름은 왼쪽에서 오른쪽으로 *format*의 형 정보와 일치합니다. 성공하면,

`PyArg_ParseTupleAndKeywords()` 는 참을 반환하고, 그렇지 않으면 거짓을 반환하고 적절한 예외를 발생시킵니다.

**참고:** 키워드 인자를 사용할 때 중첩된 튜플을 구문분석할 수 없습니다! `kwlist`에 없는 키워드 매개 변수가 전달되면 `TypeError`를 발생시킵니다.

다음은 Geoff Philbrick ([philbrick@hks.com](mailto:philbrick@hks.com)) 의 예제를 기반으로 한, 키워드를 사용하는 예제 모듈입니다:

```
#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    const char *state = "a stiff";
    const char *action = "voom";
    const char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                     &voltage, &state, &action, &type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

    Py_RETURN_NONE;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)(void (*)(void))keywdarg_parrot, METH_VARARGS | METH_
↪KEYWORDS,
    "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static struct PyModuleDef keywdargmodule = {
    PyModuleDef_HEAD_INIT,
    "keywdarg",
    NULL,
    -1,
    keywdarg_methods
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
{
    return PyModule_Create(&keywdargmodule);
}
```

## 2.1.9 임의의 값을 구축하기

이 함수는 `PyArg_ParseTuple()` 의 반대입니다. 다음과 같이 선언됩니다:

```
PyObject *Py_BuildValue(const char *format, ...);
```

`PyArg_ParseTuple()` 에서 인식되는 것과 유사한 포맷 단위 집합을 인식하지만, 인자(함수의 출력이 아니라 입력입니다)는 포인터가 아니라 그냥 값이어야 합니다. 파이썬에서 호출한 C 함수에서 반환하기에 적합한 새 파이썬 객체를 반환합니다.

`PyArg_ParseTuple()` 과의 한 가지 차이점: 후자는 첫 번째 인자가 튜플이어야 하지만(파이썬 인자 목록은 항상 내부적으로 튜플로 표현되기 때문입니다), `Py_BuildValue()` 는 항상 튜플을 빌드하지는 않습니다. 포맷 문자열에 둘 이상의 포맷 단위가 포함된 경우에만 튜플을 빌드합니다. 포맷 문자열이 비어 있으면 `None` 을 반환합니다; 정확히 하나의 포맷 단위를 포함하면, 그것이 무엇이건 해당 포맷 단위가 기술하는 객체를 반환합니다. 크기가 0 이나 1 인 튜플을 강제로 반환하도록 하려면, 포맷 문자열을 괄호로 묶으십시오.

예제 (왼쪽은 호출이고, 오른쪽은 결과 파이썬 값입니다):

<code>Py_BuildValue("")</code>	<code>None</code>
<code>Py_BuildValue("i", 123)</code>	<code>123</code>
<code>Py_BuildValue("iii", 123, 456, 789)</code>	<code>(123, 456, 789)</code>
<code>Py_BuildValue("s", "hello")</code>	<code>'hello'</code>
<code>Py_BuildValue("y", "hello")</code>	<code>b'hello'</code>
<code>Py_BuildValue("ss", "hello", "world")</code>	<code>('hello', 'world')</code>
<code>Py_BuildValue("s#", "hello", 4)</code>	<code>'hell'</code>
<code>Py_BuildValue("y#", "hello", 4)</code>	<code>b'hell'</code>
<code>Py_BuildValue("()")</code>	<code>()</code>
<code>Py_BuildValue("(i)", 123)</code>	<code>(123,)</code>
<code>Py_BuildValue("(ii)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("(i,i)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("[i,i]", 123, 456)</code>	<code>[123, 456]</code>
<code>Py_BuildValue("{s:i,s:i}",</code>	
<code>    "abc", 123, "def", 456)</code>	<code>{'abc': 123, 'def': 456}</code>
<code>Py_BuildValue("((ii)(ii))(ii)",</code>	
<code>    1, 2, 3, 4, 5, 6)</code>	<code>((1, 2), (3, 4)), (5, 6))</code>

## 2.1.10 참조 횟수

C나 C++ 와 같은 언어에서, 힙에서 메모리의 동적 할당과 할당 해제하는 것은 프로그래머가 담당합니다. C에서는, `malloc()` 과 `free()` 함수를 사용하여 이 작업을 수행합니다. C++에서는, 연산자 `new`와 `delete` 는 본질적으로 같은 의미로 사용되며 우리는 뒤따르는 논의를 C의 경우로 제한하겠습니다.

`malloc()` 으로 할당된 모든 메모리 블록은 `free()` 를 정확히 한 번 호출하여 사용 가능한 메모리 풀로 반환되어야 합니다. 적시에 `free()` 를 호출하는 것이 중요합니다. 블록의 주소를 잊어버렸지만, `free()` 를 호출하지 않으면 프로그램이 종료될 때까지 블록을 차지하는 메모리를 재사용할 수 없습니다. 이것을 메모리 누수(*memory leak*)라고 합니다. 반면에, 프로그램이 블록에 대해 `free()` 를 호출한 다음 블록을 계속 사용하면, 다른 `malloc()` 호출을 통해 블록을 재사용할 때 충돌이 발생합니다. 이것을 해제된 메모리 사용하기(*using freed memory*)라고 합니다. 초기화되지 않은 데이터를 참조하는 것과 같은 나쁜 결과를 초래합니다 — 코어 덤프, 잘못된 결과, 미스터리한 충돌.

메모리 누수의 일반적인 원인은 코드를 통한 비정상적인 경로입니다. 예를 들어, 함수는 메모리 블록을 할당하고, 어떤 계산을 한 다음, 블록을 다시 해제할 수 있습니다. 이제 함수에 대한 요구 사항이 변경되어 여러 조건을 감지하는 계산에 대한 검사를 추가하고 함수가 조기에 반환할 수 있도록 합니다. 이 조기 탈출을 수행할 때, 특히 나중에 코드에 추가될 때, 할당된 메모리 블록을 해제하는 것을 잊어버리기 쉽습니다. 이러한 누수는 일단 만들어지면 종종 오랫동안 탐지되지 않습니다: 여러 탈출은 전체 호출의 작은 부분에서만 이루어지며, 대부분의 최신 시스템에는 많은 가상 메모리가 있어서, 누수 하는 함수를 자주 사용하는 오래 실행되는 프로세스에서만 누수가 나타납니다. 따라서, 이런 종류의 에러를 코딩 규칙이나 전략을 통해 누수가 발생하지 않도록 하는 것이 중요합니다.

파이썬은 `malloc()` 과 `free()` 를 많이 사용하기 때문에, 메모리 누수와 해제된 메모리 사용을 피하는 전략이 필요합니다. 선택된 방법을 참조 횟수 세기(*reference counting*)라고 합니다. 원리는 간단합니다: 모든 객체에는 카운터를 포함합니다, 카운터는 객체에 대한 참조가 어딘가에 저장될 때 증가하고, 참조가 삭제될 때 감소합니다. 카운터가 0에 도달하면, 객체에 대한 마지막 참조가 삭제된 것이고 객체가 해제됩니다.

대체 전략을 자동 가비지 수집(*automatic garbage collection*)이라고 합니다. (때로는, 참조 횟수 세기도 가비지 수집 전략이라고 해서, 두 가지를 구별하기 위해 《자동》을 붙였습니다.) 자동 가비지 수집의 가장 큰 장점은 사용자가 `free()` 를 명시적으로 호출할 필요가 없다는 것입니다. (또 다른 주장된 이점은 속도나 메모리 사용량의 개선이지만 — 이것은 견고한 사실이 아닙니다.) 단점은 C의 경우 참조 횟수 세기는 이식성 있게 구현할 수 있지만 (함수 `malloc()` 과 `free()` 를 사용할 수 있는 한 — 이는 C 표준이 보장합니다), 실제로 이식성 있는 자동 가비지 수집기가 없다는 것입니다. 언젠가 C를 위해 충분히 이식성 있는 자동 가비지 수집기를 사용할 수 있을 것입니다. 그때까지, 우리는 참조 횟수와 함께 살아야 할 것입니다.

파이썬은 전통적인 참조 횟수 세기 구현을 사용하지만, 참조 순환을 감지하는 순환 감지기도 제공합니다. 이를 통해 응용 프로그램은 직접적이거나 간접적인 순환 참조를 만드는 것(이것이 참조 횟수만 사용하여 구현된 가비지 수집의 약점입니다)에 대해 걱정하지 않아도 됩니다. 참조 순환은 (어쩌면 간접적으로) 자신에 대한 참조를 포함하는 객체로 구성되어서, 순환의 각 객체는 0이 아닌 참조 횟수를 갖습니다. 일반적인 참조 횟수 세기 구현에서는 순환 자체에 대한 추가 참조가 없더라도 참조 순환의 객체에 속하는 메모리나 순환에 속한 객체에서 참조된 메모리를 회수할 수 없습니다.

순환 검출기는 가비지 순환을 감지하고 이를 재활용할 수 있습니다. gc 모듈은 구성 인터페이스와 실행 시간에 탐지기를 비활성화하는 기능뿐만 아니라 탐지기를 실행하는 방법(`collect()` 함수)을 제공합니다. 순환 검출기는 선택적 구성 요소로 간주합니다; 기본적으로 포함되어 있지만, 유닉스 플랫폼(맥 OS X 포함)의 **configure** 스크립트에 `--without-cycle-gc` 옵션을 사용하여 빌드 시 비활성화할 수 있습니다. 이런 방식으로 순환 탐지기를 비활성화하면 gc 모듈을 사용할 수 없습니다.

## 파이썬에서 참조 횟수 세기

참조 횟수의 증가와 감소를 처리하는 두 개의 매크로 `Py_INCREF(x)` 와 `Py_DECREF(x)` 가 있습니다. `Py_DECREF()` 는 횟수가 0에 도달하면 객체를 해제하기도 합니다. 유연성을 위해, `free()` 를 직접 호출하지 않습니다 — 대신, 객체의 형 객체(*type object*)에 있는 함수 포인터를 통해 호출합니다. 이 목적(및 기타)을 위해 모든 객체에는 해당 형 객체에 대한 포인터도 포함됩니다.

이제 큰 질문이 남습니다: 언제 `Py_INCREF(x)` 와 `Py_DECREF(x)` 를 사용합니까? 먼저 몇 가지 용어를 소개하겠습니다. 아무도 객체를 《소유(owns)》하지 않습니다! 그러나, 객체에 대한 참조를 소유(*own a reference*)할 수 있습니다. 객체의 참조 횟수는 이제 이 객체에 대한 참조를 소유한 수로 정의됩니다. 참조 소유자는 더는 참조가 필요하지 않을 때 `Py_DECREF()` 를 호출해야 합니다. 참조의 소유권을 양도할 수 있습니다. 소유한 참조를 처분하는 세 가지 방법이 있습니다: 전달, 저장 및 `Py_DECREF()` 호출. 소유한 참조를 처분하지 않으면 메모리 누수가 발생합니다.

객체에 대한 참조를 빌리는(*borrow*)<sup>2</sup> 것도 가능합니다. 참조의 대여자(*borrower*)는 `Py_DECREF()` 를 호출해서는 안 됩니다. 대여자는 빌린 소유자보다 더 오래 객체를 붙잡아서는 안 됩니다. 소유자가 처분한 후 빌린 참조를 사용하면 해제된 메모리를 사용할 위험이 있어서 절대 피해야 합니다<sup>3</sup>.

참조 소유에 비교할 때 빌리기의 이점은 코드를 통한 가능한 모든 경로에서 참조를 처리할 필요가 없다는 것입니다 — 즉, 빌려온 참조를 사용하면 조기 종료 시에 누수의 위험이 없습니다. 소유하는 것에 비해 빌리는 것의 단점은, 결보기에는 올바른 코드지만, 빌려준 소유자가 실제로는 참조를 처분한 후에 빌린 참조가 사용될 수 있는 미묘한 상황이 있다는 것입니다.

빌린 참조는 `Py_INCREF()` 를 호출하여 소유한 참조로 변경할 수 있습니다. 이는 참조를 빌려온 소유자의 상태에 영향을 미치지 않습니다 — 새로운 소유된 참조를 만들고, 완전한 소유자 책임을 부여합니다(이전 소유자뿐만 아니라, 새 소유자는 참조를 올바르게 처분해야 합니다).

<sup>2</sup> 참조 《빌리기(borrowing)》은 유는 완전히 올바르지 않습니다: 소유자는 여전히 참조 사본을 가지고 있습니다.

<sup>3</sup> 참조 횟수가 1 이상인지 확인하는 것은 작동하지 않습니다 — 참조 횟수 자체가 해제된 메모리에 있을 수 있어서 다른 객체에 재사용될 수 있습니다!

## 소유권 규칙

객체 참조가 함수 안팎으로 전달될 때마다, 소유권이 참조와 함께 전달되는지 그렇지 않은지는 함수 인터페이스 명세의 일부입니다.

객체에 대한 참조를 반환하는 대부분의 함수는 참조와 함께 소유권을 전달합니다. 특히, `PyLong_FromLong()` 이나 `Py_BuildValue()` 와 같은 새 객체를 만드는 기능을 가진 모든 함수는 소유권을 수신자에게 전달합니다. 객체가 실제로 새 객체가 아니더라도, 여전히 해당 객체에 대한 새 참조의 소유권을 받습니다. 예를 들어, `PyLong_FromLong()` 은 흔히 사용되는 값의 캐시를 유지하고 캐시된 항목에 대한 참조를 반환할 수 있습니다.

다른 객체에서 객체를 추출하는 많은 함수도 참조와 함께 소유권을 전달합니다, 예를 들어 `PyObject_GetAttrString()`. 그러나 몇 가지 일반적인 루틴이 예외이기 때문에 그림이 명확하지 않습니다: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()` 및 `PyDict_GetItemString()` 은 모두 튜플, 리스트 또는 딕셔너리에서 빌린 참조를 반환합니다.

`PyImport_AddModule()` 함수도 실제로는 반환하는 객체를 만들 수 있지만 빌린 참조를 반환합니다: 객체에 대한 소유한 참조가 `sys.modules`에 저장되어 있기 때문에 가능합니다.

객체 참조를 다른 함수에 전달할 때, 일반적으로, 함수는 여러분으로부터 참조를 빌립니다 — 참조를 저장해야 하면, `Py_INCREF()` 를 사용하여 독립 소유자가 됩니다. 이 규칙에는 두 가지 중요한 예외가 있습니다: `PyTuple_SetItem()` 과 `PyList_SetItem()`. 이 함수들은 전달된 항목에 대한 소유권을 취합니다 — 설사 실패하더라도! (`PyDict_SetItem()` 과 그 친구들은 소유권을 취하지 않습니다 — 이들은 《정상》입니다.)

C 함수가 파이썬에서 호출될 때, 호출자로부터 온 인자에 대한 참조를 빌립니다. 호출자는 객체에 대한 참조를 소유하기 때문에, 빌린 참조의 수명은 함수가 반환될 때까지 보장됩니다. 이러한 빌린 참조를 저장하거나 전달해야 할 때만, `Py_INCREF()` 를 호출하여 소유한 참조로 만들어야 합니다.

파이썬에서 호출된 C 함수에서 반환된 객체 참조는 소유한 참조여야 합니다 — 소유권은 함수에서 호출자로 전달됩니다.

## 살얼음

겉보기에 무해한 빌린 참조의 사용이 문제를 일으킬 수 있는 몇 가지 상황이 있습니다. 이것들은 모두 참조의 소유자가 참조를 처분하도록 할 수 있는 인터프리터의 묵시적 호출과 관련이 있습니다.

가장 먼저 알아야 할 가장 중요한 경우는 리스트 항목에 대한 참조를 빌리는 동안 관련이 없는 객체에서 `Py_DECREF()` 를 사용하는 것입니다. 예를 들어:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

이 함수는 먼저 `list[0]`에 대한 참조를 빌린 다음, `list[1]`을 값 0으로 바꾸고, 마지막으로 빌린 참조를 인쇄합니다. 무해해 보이지요? 하지만 그렇지 않습니다!

`PyList_SetItem()` 으로의 제어 흐름을 따라가 봅시다. 리스트는 모든 항목에 대한 참조를 소유해서, 항목 1을 교체할 때 원래 항목 1을 처분(dispose)해야 합니다. 이제 원본 항목 1이 사용자 정의 클래스의 인스턴스라고 가정하고, 클래스가 `__del__()` 메서드를 정의했다고 더 가정해 봅시다. 이 클래스 인스턴스의 참조 횟수가 1일 때, 이를 처분하면 `__del__()` 메서드가 호출됩니다.

파이썬으로 작성되었기 때문에, `__del__()` 메서드는 임의의 파이썬 코드를 실행할 수 있습니다. 그것이 `bug()`에서 `item`에 대한 참조를 무효로 하는 작업을 수행할 수 있을까요? 물론입니다! `bug()`에 전달된 리스트가 `__del__()` 메서드에서 액세스 가능하다고 가정하면, `del list[0]`의 효과를 주는 문장을 실행할 수 있으며, 이것이 해당 객체에 대한 마지막 참조라고 가정하면, 그것과 연관된 메모리를 해제하고, 그래서 `item`을 무효로 합니다.



문제의 원인을 알고 나면, 해결 방법은 쉽습니다: 일시적으로 참조 횟수를 늘리십시오. 올바른 버전의 함수는 다음과 같습니다:

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

이것은 실제 이야기입니다. 이전 버전의 파이썬에는 이 버그의 변형이 포함되어 있으며 누군가 `__del__()` 메시지가 실패하는 이유를 알아내기 위해 C 디버거에서 상당한 시간을 보냈습니다...

빌린 참조에 문제가 있는 두 번째 경우는 스레드와 관련된 변형입니다. 일반적으로, 파이썬의 전체 객체 공간을 보호하는 전역 록이 있어서, 파이썬 인터프리터의 여러 스레드는 다른 것들의 길에 끼어들 수 없습니다. 그러나, 매크로 `Py_BEGIN_ALLOW_THREADS`를 사용하여 이 록을 일시적으로 해제하고 `Py_END_ALLOW_THREADS`를 사용하여 다시 확보할 수 있습니다. 이는 블로킹 I/O 호출에서 흔한데, I/O가 완료되기를 기다리는 동안 다른 스레드가 프로세서를 사용할 수 있도록 합니다. 분명히, 다음 함수는 이전 함수와 같은 문제가 있습니다:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

## NULL 포인터

일반적으로, 객체 참조를 인자로 취하는 함수는 NULL 포인터를 전달할 것으로 기대하지 않으며, 그렇게 하면 코어를 덤프합니다(또는 이후의 코어 덤프를 유발합니다). 객체 참조를 반환하는 함수는 일반적으로 예외가 발생했음을 나타내기 위해서만 NULL을 반환합니다. NULL 인자를 검사하지 않는 이유는 함수들이 종종 자신이 받은 객체를 다른 함수에 전달하기 때문입니다 — 각 함수가 NULL을 검사한다면, 중복 검사가 많이 발생하고 코드가 더 느리게 실행됩니다.

NULL일 수 있는 포인터가 수신될 때 《소스》에서만 NULL을 검사하는 것이 좋습니다, 예를 들어, `malloc()`이나 예외를 발생시킬 수 있는 함수에서.

매크로 `Py_INCREF()`와 `Py_DECREF()`는 NULL 포인터를 검사하지 않습니다 — 하지만, 그들의 변형 `Py_XINCRF()`와 `Py_XDECREF()`는 확인합니다.

특정 객체 형을 확인하기 위한 매크로(`Pytype_Check()`)는 NULL 포인터를 확인하지 않습니다 — 다시, 여러 기대하는 형에 대해 객체를 검사하기 위해 연속해서 이들을 여러 번 호출하는 코드가 많아서, 중복 검사가 생성됩니다. NULL 검사를 하는 변형은 없습니다.

C 함수 호출 메커니즘은 C 함수에 전달된 인자 목록(예에서는 `args`)이 절대 NULL이 아님을 보장합니다 — 실제로는 항상 튜플임을 보장합니다<sup>4</sup>.

NULL 포인터를 파이썬 사용자에게 《빠져나가게》 만드는 것은 심각한 에러입니다.

<sup>4</sup> 《오래된》 스타일 호출 규칙을 사용할 때 이러한 보장은 유지되지 않습니다 — 이것은 여전히 기존 코드에서 많이 발견됩니다.

## 2.1.11 C++로 확장 작성하기

C++로 확장 모듈을 작성할 수 있습니다. 일부 제한 사항이 적용됩니다. 메인 프로그램(파이썬 인터프리터)이 C 컴파일러로 컴파일되고 링크되면, 생성자가 있는 전역이나 정적(static) 객체를 사용할 수 없습니다. 메인 프로그램이 C++ 컴파일러로 링크된 경우에는 문제가 되지 않습니다. 파이썬 인터프리터가 호출할 함수(특히, 모듈 초기화 함수)는 `extern "C"`를 사용하여 선언해야 합니다. `extern "C" {...}`로 파이썬 헤더 파일을 묶을 필요는 없습니다—`__cplusplus` 기호가 정의되면(모든 최신 C++ 컴파일러가 이 기호를 정의합니다) 이미 이 형식을 사용합니다.

## 2.1.12 확장 모듈을 위한 C API 제공하기

많은 확장 모듈은 단지 파이썬에서 사용할 새로운 함수와 형을 제공하지만, 때로 확장 모듈의 코드가 다른 확장 모듈에 유용할 수 있습니다. 예를 들어, 확장 모듈은 순서 없는 리스트처럼 작동하는 《컬렉션》 형을 구현할 수 있습니다. 표준 파이썬 리스트 형에 확장 모듈이 리스트를 만들고 조작할 수 있게 하는 C API가 있는 것처럼, 이 새로운 컬렉션 형에는 다른 확장 모듈에서 직접 조작할 수 있는 C 함수 집합이 있어야 합니다.

첫눈에 이것은 쉬운 것처럼 보입니다; 단지 함수를 작성하고(물론 `static`을 선언하지 않고), 적절한 헤더 파일을 제공하고, C API를 설명합니다. 사실 이것은 모든 확장 모듈이 항상 파이썬 인터프리터와 정적으로 링크되어 있다면 작동합니다. 그러나 모듈을 공유 라이브러리로 사용하면, 한 모듈에 정의된 기호가 다른 모듈에서 보이지 않을 수 있습니다. 가시성의 세부 사항은 운영 체제에 따라 다릅니다; 어떤 시스템은 파이썬 인터프리터와 모든 확장 모듈에 하나의 전역 이름 공간을 사용하는 반면(예를 들어 윈도우), 다른 시스템은 모듈 링크 시점에 임포트 되는 기호의 목록을 명시적으로 요구하거나(AIX가 하나의 예입니다), 여러 전략 중 선택할 수 있도록 합니다(대부분의 유닉스). 또한 기호가 전역적으로 보이더라도, 호출하려는 함수를 가진 모듈이 아직 로드되지 않았을 수 있습니다!

따라서 이식성에는 기호 가시성에 대해 가정하지 않을 것이 요구됩니다. 이것은 다른 확장 모듈과의 이름 충돌을 피하고자, 모듈의 초기화 함수를 제외한 확장 모듈의 모든 기호를 `static`으로 선언해야 함을 의미합니다(섹션 [모듈의 메서드 테이블과 초기화 함수](#)에서 설명되듯이). 그리고 이는 다른 확장 모듈에서 액세스 해야만 하는 기호를 다른 방식으로 노출해야 함을 의미합니다.

파이썬은 한 확장 모듈에서 다른 확장 모듈로 C 수준 정보(포인터)를 전달하는 특별한 메커니즘을 제공합니다: 캡슐(Capsule). 캡슐은 포인터(`void *`)를 저장하는 파이썬 데이터형입니다. 캡슐은 C API를 통해서만 만들고 액세스할 수 있지만, 다른 파이썬 객체처럼 전달할 수 있습니다. 특히, 확장 모듈의 이름 공간에서 이름에 대입할 수 있습니다. 다른 확장 모듈은 이 모듈을 임포트 해서, 이 이름의 값을 가져온 다음, 캡슐에서 포인터를 가져올 수 있습니다.

확장 모듈의 C API를 노출하는 데 캡슐을 사용하는 방법에는 여러 가지가 있습니다. 각 함수가 자신만의 캡슐을 얻거나, 모든 C API 포인터가 저장된 배열의 주소를 캡슐로 게시할 수 있습니다. 그리고 포인터를 저장하고 꺼내는 다양한 작업은 코드를 제공하는 모듈과 클라이언트 모듈 간에 여러 방식으로 분산될 수 있습니다.

어떤 방법을 선택하든, 캡슐 이름을 올바르게 지정하는 것이 중요합니다. `PyCapsule_New()` 함수는 `name` 매개 변수(`const char *`)를 취합니다; `NULL name`을 전달할 수는 있지만, 이름을 지정하도록 강력히 권고합니다. 적절하게 이름 붙인 캡슐은 어느 정도의 실행 시간 형 안전성을 제공합니다; 하나의 이름 없는 캡슐을 다른 캡슐과 구별할 수 있는 적절한 방법은 없습니다.

특히, C API를 공개하는 데 사용되는 캡슐에는 다음 규칙에 따라 이름을 지정해야 합니다:

```
modulename.attributename
```

편의 함수 `PyCapsule_Import()`를 사용하면 캡슐을 통해 제공된 C API를 쉽게 로드 할 수 있지만, 캡슐 이름이 이 규칙과 일치할 때만 그렇습니다. 이 동작은 C API 사용자에게 자신이 로드 한 캡슐에 올바른 C API가 포함되어 있다는 확신을 줍니다.

다음 예제는 대부분의 부담을 내보내는 모듈의 작성자에게 주는 방식을 보여주는데, 일반적으로 사용되는 라이브러리 모듈에 적합합니다. 캡슐의 값이 되는 `void` 포인터의 배열에 모든 C API 포인터(이 예에서는 하나뿐입니다!)를 저장합니다. 모듈에 해당하는 헤더 파일은 모듈을 임포트 하고 C API 포인터를 가져오는 매크로를 제공합니다; 클라이언트 모듈은 C API에 액세스하기 전에 이 매크로를 호출하기만 하면 됩니다.



내보내는 모듈은 섹션 [간단한 예](#)의 spam 모듈을 수정한 것입니다. spam.system() 함수는 C 라이브러리 함수 system()을 직접 호출하지는 않고, 실제로는 더 복잡한 작업을 수행하는 (가령 모든 명령에 《spam》을 추가하는 것과 같은) PySpam\_System() 함수를 호출합니다. 이 함수 PySpam\_System()도 다른 확장 모듈로 내보냅니다.

함수 PySpam\_System()은 평범한 C 함수이며, 다른 모든 것과 같이 static으로 선언되었습니다:

```
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

spam\_system() 함수는 사소하게 수정됩니다:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return PyLong_FromLong(sts);
}
```

모듈의 시작 부분에서, 다음 줄 바로 다음에

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

다음 두 줄을 더 추가해야 합니다:

```
#define SPAM_MODULE
#include "spammodule.h"
```

#define은 헤더 파일이 클라이언트 모듈이 아닌 내보내는 모듈에 포함됨을 알리는 데 사용됩니다. 마지막으로, 모듈의 초기화 함수는 C API 포인터 배열을 초기화해야 합니다:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's address */
    c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API", NULL);

    if (PyModule_AddObject(m, "_C_API", c_api_object) < 0) {
        Py_XDECREF(c_api_object);
        Py_DECREF(m);
        return NULL;
    }
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    return m;
}

```

PySpam\_API는 static으로 선언됩니다; 그렇지 않으면 PyInit\_spam() 이 종료할 때 포인터 배열이 사라집니다!

작업 대부분은 헤더 파일 spammodule.h에 있으며, 다음과 같습니다:

```

#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1

#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
    (*(PySpam_System_RETURN (*)(PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* Return -1 on error, 0 on success.
 * PyCapsule_Import will set an exception if there's an error.
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */

```

PySpam\_System() 함수에 액세스하기 위해 클라이언트 모듈이 해야 할 일은 초기화 함수에서 함수(사실 매크로) import\_spam() 을 호출하는 것이 전부입니다:

```

PyMODINIT_FUNC
PyInit_client(void)

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

{
    PyObject *m;

    m = PyModule_Create(&clientmodule);
    if (m == NULL)
        return NULL;
    if (import_spam() < 0)
        return NULL;
    /* additional initialization can happen here */
    return m;
}

```

이 방법의 주요 단점은 파일 spammodule.h가 다소 복잡하다는 것입니다. 그러나, 기본 구조는 내보내는 함수마다 같아서, 한 번만 학습하면 됩니다.

마지막으로 캡슐은 추가 기능을 제공하며, 특히 캡슐에 저장된 포인터의 메모리 할당과 할당 해제에 유용합니다. 세부 사항은 파이썬/C API 레퍼런스 매뉴얼의 **capsules** 섹션과 캡슐 구현(파이썬 소스 코드 배포의 Include/pycapsule.h와 Objects/pycapsule.c 파일)에 설명되어 있습니다.

## 2.2 확장형 정의하기: 자습서

파이썬은 C 확장 모듈 작성자가 내장 str과 list 형과 마찬가지로 파이썬 코드에서 조작할 수 있는 새로운 형을 정의할 수 있도록 합니다. 모든 확장형의 코드는 패턴을 따르지만, 시작하기 전에 이해해야 할 세부 사항이 있습니다. 이 설명서는 주제에 대한 간단한 소개입니다.

### 2.2.1 기초

CPython 런타임은 모든 파이썬 객체를 PyObject\* 형의 변수로 간주하는데, 이는 모든 파이썬 객체의 《베이스형》 역할을 합니다. PyObject 구조체 자체는 객체의 참조 횟수와 객체의 《형 객체》에 대한 포인터만 포함합니다. 여기가 액션이 일어나는 곳입니다; 형 객체는 예를 들어 객체에서 어트리뷰트를 조회하거나, 메서드를 호출하거나, 다른 객체와 곱할 때 인터프리터가 호출하는 (C) 함수를 결정합니다. 이러한 C 함수를 《형 메서드》라고 합니다.

따라서, 새 확장형을 정의하려면, 새 형 객체를 만들어야 합니다.

이런 종류의 것은 예제로만 설명할 수 있어서, 여기에 C 확장 모듈 custom 내에서 Custom이라는 새 형을 정의하는 최소한이지만 완전한 모듈이 있습니다:

---

**참고:** 여기에 표시하는 것은 정적인 (static) 확장형을 정의하는 전통적인 방법입니다. 대부분의 용도에 적합해야 합니다. C API는 또한 PyType\_FromSpec() 함수를 사용하여 힙 할당 확장형을 정의할 수 있습니다만, 이 자습서에서는 다루지 않습니다.

---

```

#define PY_SSIZE_T_CLEAN
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} CustomObject;

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

        .tp_itemsize = 0,
        .tp_flags = Py_TPFLAGS_DEFAULT,
        .tp_new = PyType_GenericNew,
    };

    static PyModuleDef custommodule = {
        PyModuleDef_HEAD_INIT,
        .m_name = "custom",
        .m_doc = "Example module that creates an extension type.",
        .m_size = -1,
    };

    PyMODINIT_FUNC
    PyInit_custom(void)
    {
        PyObject *m;
        if (PyType_Ready(&CustomType) < 0)
            return NULL;

        m = PyModule_Create(&custommodule);
        if (m == NULL)
            return NULL;

        Py_INCREF(&CustomType);
        if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
            Py_DECREF(&CustomType);
            Py_DECREF(m);
            return NULL;
        }

        return m;
    }
}
    
```

이제는 한 번에 배워야 할 것이 많지만, 이전 장과 비슷해 보이기를 바랍니다. 이 파일은 세 가지를 정의합니다:

1. Custom 객체에 포함된 것: CustomObject 구조체이며, 각 Custom 인스턴스마다 한 번씩 할당됩니다.
2. Custom 형의 작동 방식: CustomType 구조체이며, 특정 연산이 요청될 때 인터프리터가 검사하는 플래그와 함수 포인터 집합을 정의합니다.
3. custom 모듈을 초기화하는 방법: PyInit\_custom 함수와 관련 custommodule 구조체입니다.

첫 번째 것은:

```

typedef struct {
    PyObject_HEAD
} CustomObject;
    
```

이것이 Custom 객체에 포함될 것입니다. PyObject\_HEAD는 각 객체 구조체의 시작 부분에 필수적으로 오는 것이며, PyObject 형의 ob\_base라는 필드를 정의하여, 형 객체에 대한 포인터와 참조 횟수를 포함합니다(이것들은 각각 매크로 Py\_TYPE과 Py\_REFCNT를 사용하여 액세스 할 수 있습니다). 이것이 매크로인 이유는 배치(layout)를 추상화하고 디버그 빌드에서 추가 필드를 활성화하기 위한 것입니다.

**참고:** PyObject\_HEAD 매크로 뒤에는 세미콜론이 없습니다. 실수로 추가하는 것에 주의하십시오: 일부 컴파일러는 불평할 것입니다.

물론, 객체는 일반적으로 표준 PyObject\_HEAD 관용구 외에 추가 데이터를 저장합니다; 예를 들어, 표준 파이썬 floats에 대한 정의는 다음과 같습니다:

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

두 번째 것은 형 객체의 정의입니다.

```
static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};
```

**참고:** 신경 쓰지 않는 모든 PyTypeObject 필드를 나열하지 않고 필드의 선언 순서를 신경 쓰지 않으려면, 위와 같이 C99 스타일의 지명 (designated) 초기화자를 사용하는 것이 좋습니다.

object.h에 있는 PyTypeObject의 실제 정의는 위의 정의보다 더 많은 필드를 갖습니다. 나머지 필드는 C 컴파일러에 의해 0으로 채워지며, 필요하지 않으면 명시적으로 지정하지 않는 것이 일반적입니다.

한 번에 한 필드씩 따로 다루려고 합니다:

```
PyVarObject_HEAD_INIT(NULL, 0)
```

이 줄은 위에서 언급한 ob\_base 필드를 초기화하기 위한 필수 상용구입니다.

```
.tp_name = "custom.Custom",
```

우리 형의 이름. 이것은 객체의 기본 텍스트 표현과 일부 에러 메시지에 나타납니다, 예를 들어:

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Custom") to str
```

이름이 모듈 이름과 모듈 내 형의 이름을 모두 포함하는 점으로 구분된 이름임에 유의하십시오. 이 경우 모듈은 custom이고 형은 Custom이라서, 형 이름을 custom.Custom으로 설정합니다. 형이 pydoc과 pickle 모듈과 호환되도록 하려면 실제 점으로 구분된 импорт 경로를 사용하는 것이 중요합니다.

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

이것은 새로운 Custom 인스턴스를 만들 때 파이썬이 할당할 메모리량을 알 수 있도록 하기 위한 것입니다. tp\_itemsize는 가변 크기 객체에만 사용되며 그렇지 않으면 0이어야 합니다.

**참고:** 파이썬에서 형을 서브클래싱 할 수 있기를 원하고, 형이 베이스형과 같은 tp\_basicsize를 가지면, 다중 상속에 문제가 있을 수 있습니다. 형의 파이썬 서브 클래스는 \_\_bases\_\_에 이 형을 먼저 나열해야 합니다, 그렇지 않으면 에러 없이 형의 \_\_new\_\_() 메서드를 호출할 수 없습니다. 형이 베이스형보다 큰 tp\_basicsize 값을 갖도록 하여 이 문제점을 피할 수 있습니다. 대부분의 경우, 이것은 어쨌든 만족하는데, 베이스형이 object이거나, 그렇지 않으면 베이스형에 데이터 멤버를 추가하여 크기를 늘리기 때문입니다.

클래스 플래그를 Py\_TPFLAGS\_DEFAULT로 설정합니다.

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

모든 형은 이 상수를 플래그에 포함해야 합니다. 적어도 파이썬 3.3까지 정의된 모든 멤버를 활성화합니다. 추가 멤버가 필요하다면, 해당 플래그를 OR 해야 합니다.

tp\_doc에 형의 독스트링을 제공합니다.

```
.tp_doc = "Custom objects",
```

객체 생성을 가능하게 하려면, tp\_new 처리기를 제공해야 합니다. 이것은 파이썬 메서드 \_\_new\_\_()와 동등하지만, 명시적으로 지정해야 합니다. 이 경우에는, API 함수 PyType\_GenericNew()에서 제공하는 기본 구현을 그냥 사용할 수 있습니다.

```
.tp_new = PyType_GenericNew,
```

PyInit\_custom()의 일부 코드를 제외하고, 파일의 다른 모든 내용은 익숙해야 합니다:

```
if (PyType_Ready(&CustomType) < 0)
    return;
```

이것은 Custom 형을 초기화하는데, 처음에 NULL로 설정한 ob\_type을 포함하여, 여러 멤버를 적절한 기본값으로 채웁니다.

```
Py_INCREF(&CustomType);
if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
    Py_DECREF(&CustomType);
    Py_DECREF(m);
    return NULL;
}
```

이것은 형을 모듈 디렉터리에 추가합니다. Custom 클래스를 호출하여 Custom 인스턴스를 만들 수 있도록 합니다:

```
>>> import custom
>>> mycustom = custom.Custom()
```

이게 전부입니다! 남아있는 것은 빌드하는 것입니다; 위의 코드를 custom.c라는 파일에 넣고:

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[Extension("custom", ["custom.c"])])
```

를 setup.py라는 파일에 넣은 다음; 다음을

```
$ python setup.py build
```

셸에서 입력하면 서브 디렉터리에 파일 custom.so를 생성해야 합니다; 해당 디렉터리로 이동하여 파이썬을 시작하십시오 — import custom 할 수 있고 Custom 객체로 놀 수 있습니다.

그렇게 어렵지 않습니다, 그렇지 않나요?

물론, 현재 Custom 형은 그리 흥미롭지 않습니다. 데이터가 없고 아무것도 하지 않습니다. 서브 클래스싱조차 할 수 없습니다.

**참고:** 이 설명서는 C 확장을 빌드하기 위한 표준 distutils 모듈을 보여 주지만, 실제 사용 사례에서는 새롭고 유지 관리가 잘 된 setuptools 라이브러리를 사용하는 것이 좋습니다. 이 작업을 수행하는 방법에 대한 설명서는 이 문서의 범위를 넘어서고 [파이썬 패키징 사용자 지침서](#)에서 찾을 수 있습니다.

## 2.2.2 기초 예제에 데이터와 메서드 추가하기

데이터와 메서드를 추가하도록 기초 예제를 확장해 봅시다. 형을 베이스 클래스로도 사용할 수 있도록 합시다. 다음 기능을 추가하는 새 모듈 custom2를 만들 것입니다:

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom2",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};
    
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```
PyMODINIT_FUNC
PyInit_custom2(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

이 버전의 모듈에는 여러 가지 변경 사항이 있습니다.

다음과 같은 추가 포함(include)을 추가했습니다:

```
#include <structmember.h>
```

이 포함은 나중에 설명하는 것처럼 어트리뷰트를 처리하는 데 사용하는 선언을 제공합니다.

Custom 형은 이제 C 구조체에 *first*, *last* 및 *number*의 세 가지 데이터 어트리뷰트가 있습니다. *first*와 *last* 변수는 이름과 성을 포함하는 파이썬 문자열입니다. *number* 어트리뷰트는 C 정수입니다.

객체 구조체는 다음과 같이 갱신됩니다:

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;
```

이제 관리할 데이터가 있기 때문에, 객체 할당과 할당 해제에 관해 더욱 신중해야 합니다. 최소한, 할당 해제 메서드가 필요합니다:

```
static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

이는 tp\_dealloc 멤버에 대입됩니다:

```
.tp_dealloc = (destructor) Custom_dealloc,
```

이 메서드는 먼저 두 파이썬 어트리뷰트의 참조 횟수를 지웁니다. Py\_XDECREF()는 인자가 NULL인 경우(tp\_new가 중간에 실패하면 발생할 수 있습니다)를 올바르게 처리합니다. 그런 다음 객체 형(Py\_TYPE(self)로 계산합니다)의 tp\_free 멤버를 호출하여 객체의 메모리를 해제합니다. 객체 형이 CustomType이 아닐 수 있음에 유의하십시오, 객체는 서브 클래스의 인스턴스일 수 있기 때문입니다.

참고: CustomObject \* 인자를 취하도록 Custom\_dealloc을 정의했지만, tp\_dealloc 함수 포인터

는 `PyObject *` 인자를 받을 것으로 기대하기 때문에 위의 `destructor`로의 명시적 캐스트가 필요합니다. 그렇지 않으면, 컴파일러에서 경고가 발생합니다. 이것이 C로 하는 객체 지향 다형성입니다!

우리는 성과 이름이 빈 문자열로 초기화되도록 하고 싶어서, `tp_new` 구현을 제공합니다:

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}
```

그리고 그것을 `tp_new` 멤버에 설치합니다:

```
.tp_new = Custom_new,
```

`tp_new` 처리기는 형의 객체를 (초기화와 대비하여) 생성하는 책임을 집니다. 파이썬에서 `__new__()` 메서드로 노출됩니다. `tp_new` 멤버를 정의할 필요는 없으며, 실제로 많은 확장형은 위의 `Custom` 형의 첫 번째 버전에서처럼 `PyType_GenericNew()` 를 재사용하기만 합니다. 지금은, `tp_new` 처리기를 사용하여 `first`와 `last` 어트리뷰트를 `NULL`이 아닌 기본값으로 초기화합니다.

`tp_new`는 인스턴스화 되는 형 (서브 클래스가 인스턴스화 되면, 반드시 `CustomType`일 필요는 없습니다)과 형이 호출될 때 전달된 모든 인자가 전달되며, 만들어진 인스턴스를 반환할 것으로 기대됩니다. `tp_new` 처리기는 항상 위치와 키워드 인자를 받아들이지만, 종종 인자를 무시하고 인자 처리를 초기화 (C의 `tp_init`나 파이썬의 `__init__`) 메서드에게 남겨둡니다.

**참고:** 인터프리터가 직접 할 것이라서, `tp_new`는 명시적으로 `tp_init`를 호출하면 안 됩니다.

`tp_new` 구현은 `tp_alloc` 슬롯을 호출하여 메모리를 할당합니다:

```
self = (CustomObject *) type->tp_alloc(type, 0);
```

메모리 할당이 실패할 수 있어서, 진행하기 전에 `tp_alloc` 결과가 `NULL`이 아닌지 확인해야 합니다.

**참고:** 우리는 `tp_alloc` 슬롯을 직접 채우지 않았습니다. 대신 `PyType_Ready()` 가 베이스 클래스 (기본적으로 `object`입니다)에서 상속하여 이를 채웁니다. 대부분의 형은 기본 할당 전략을 사용합니다.

**참고:** 협업 `tp_new`(베이스형의 `tp_new`나 `__new__()` 를 호출하는 것)를 만드는 경우, 실행 시간에 메서드 결정 순서를 사용하여 호출할 메서드를 결정하려고 하지 않아야 합니다. 항상 어떤 형을 호출할지 정적으로 결정하고, 그것의 `tp_new`를 직접, 또는 `type->tp_base->tp_new`를 통해 호출하십시오. 이렇게 하지 않으면, 다른 파이썬 정의 클래스도 상속하는 여러 형의 파이썬 서브 클래스가 올바르게 작동하지 않을 수 있습니다. (특히, `TypeError`를 얻지 않으면서, 이러한 서브 클래스의 인스턴스를 만들지 못할 수도 있습니다.)

인스턴스의 초기값을 제공하는 인자를 받아들이는 초기화 함수도 정의합니다:

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}
```

이것으로 tp\_init 슬롯을 채웁니다:

```
.tp_init = (initproc) Custom_init,
```

tp\_init 슬롯은 파이썬에서 \_\_init\_\_() 메서드로 노출됩니다. 객체가 만들어진 후 초기화하는 데 사용됩니다. 초기화자는 항상 위치와 키워드 인자를 받아들이며 성공 시 0 또는 에러 시 -1을 반환해야 합니다.

tp\_new 처리기와 달리, tp\_init가 아예 호출되지 않을 수도 있습니다(예를 들어, pickle 모듈은 기본적으로 역 피콜된 인스턴스에서 \_\_init\_\_()를 호출하지 않습니다). 여러 번 호출될 수도 있습니다. 누구나 우리 객체의 \_\_init\_\_() 메서드를 호출할 수 있습니다. 이런 이유로, 새 어트리뷰트 값을 대입할 때는 각별히 주의해야 합니다. 예를 들어 first 멤버를 다음과 같이 대입하려고 할 수 있습니다:

```
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

하지만 이것은 위험합니다. 우리 형은 first 멤버의 형을 제한하지 않아서, 모든 종류의 객체가 될 수 있습니다. first 멤버에 액세스하려고 시도하는 코드가 실행되도록 하는 파괴자가 있을 수 있습니다; 또는 파괴자가 전역 인터프리터 록을 해제하고 다른 스레드에서 객체에 액세스하고 수정하는 임의의 코드가 실행되도록 할 수 있습니다.

편집증적이 되고 이 가능성으로부터 우리 자신을 보호하기 위해, 우리는 거의 항상 참조 횟수를 줄이기 전에 멤버를 다시 대입합니다. 언제 이렇게 하지 않아도 될까요?

- 참조 횟수가 1보다 크다는 것을 확실히 알고 있을 때;
- 객체의 할당 해제가 GIL을 해제하지도 않고 형의 코드를 다시 호출하지도 않음을 알고 있을 때<sup>1</sup>;
- 순환 가비지 수거를 지원하지 않는 형의 tp\_dealloc 처리기에서 참조 횟수를 감소시킬 때<sup>2</sup>.

<sup>1</sup> 이것은 객체가 문자열이나 부동 소수점과 같은 기본형이라는 것을 알고 있을 때 참입니다.

<sup>2</sup> 우리의 형이 가비지 수거를 지원하지 않기 때문에, 이 예제에서는 tp\_dealloc 처리기에서 이것을 사용했습니다.

인스턴스 변수를 어트리뷰트로 노출하려고 합니다. 이를 수행하는 방법에는 여러 가지가 있습니다. 가장 간단한 방법은 멤버 정의를 정의하는 것입니다:

```
static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};
```

그리고 `tp_members` 슬롯에 정의를 넣습니다:

```
.tp_members = Custom_members,
```

각 멤버 정의에는 멤버 이름, 형, 오프셋, 액세스 플래그 및 독스트링이 있습니다. 자세한 내용은 아래 **범용 어트리뷰트 관리** 섹션을 참조하십시오.

이 접근법의 단점은 파이썬 어트리뷰트에 대입할 수 있는 객체의 형을 제한할 방법을 제공하지 않는다는 것입니다. 이름과 성은 문자열일 것으로 기대하지만, 모든 파이썬 객체를 할당할 수 있습니다. 또한 어트리뷰트를 삭제할 수 있습니다, C 포인터를 NULL로 설정합니다. NULL이 아닌 값으로 멤버를 초기화 할 수 있지만, 어트리뷰트를 삭제하면 멤버를 NULL로 설정할 수 있습니다.

이름과 성을 이어붙여 객체 이름으로 출력하는 단일 메서드 `Custom.name()` 을 정의합니다.

```
static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

이 메서드는 `Custom`(또는 `Custom` 서브 클래스) 인스턴스를 첫 번째 인자로 취하는 C 함수로 구현됩니다. 메서드는 항상 인스턴스를 첫 번째 인자로 취합니다. 메서드는 종종 위치와 키워드 인자도 취하지만, 이 경우에는 아무것도 취하지 않아서 위치 인자 튜플이나 키워드 인자 딕셔너리를 받아들이 필요 없습니다. 이 메서드는 다음과 같은 파이썬 메서드와 동등합니다:

```
def name(self):
    return "%s %s" % (self.first, self.last)
```

`first`와 `last` 멤버가 NULL일 가능성을 확인해야 함에 유의하십시오. 삭제할 수 있기 때문인데, 이때 NULL로 설정됩니다. 이러한 어트리뷰트의 삭제를 방지하고 어트리뷰트 값을 문자열로 제한하는 것이 더 좋습니다. 다음 섹션에서 이를 수행하는 방법을 살펴보겠습니다.

이제 메서드를 정의했습니다, 메서드 정의 배열을 만들어야 합니다:

```
static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};
```

(우리는 METH\_NOARGS 플래그를 사용하여 메서드가 *self* 이외의 인자를 기대하지 않음을 나타냈음에 유의하십시오)

그리고 `tp_methods` 슬롯에 대입합니다:

```
.tp_methods = Custom_methods,
```

마지막으로, 우리의 형을 서브 클래스의 베이스 클래스로 사용할 수 있게 만들 것입니다. 우리는 지금까지 만들어지거나 사용되고 있는 객체의 형에 대해 가정을 하지 않도록 메서드를 주의해서 작성했습니다, 그래서 클래스 플래그 정의에 `Py_TPFLAGS_BASETYPE`을 추가하기만 하면 됩니다.

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
```

`PyInit_custom()`의 이름을 `PyInit_custom2()`로 바꾸고, `PyModuleDef` 구조체에서 모듈 이름을 갱신하고, `PyTypeObject` 구조체에서 전체 클래스 이름을 갱신합니다.

마지막으로, 새 모듈을 빌드하기 위해 `setup.py` 파일을 갱신합니다:

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[
          Extension("custom", ["custom.c"]),
          Extension("custom2", ["custom2.c"]),
      ])

```

## 2.2.3 데이터 어트리뷰트를 더 세밀하게 제어하기

이 섹션에서는, `Custom` 예제에서 `first`와 `last` 어트리뷰트가 설정되는 방식을 더 세밀하게 제어합니다. 이전 버전의 모듈에서는, 인스턴스 변수 `first`와 `last`를 문자열이 아닌 값으로 설정하거나 삭제할 수도 있습니다. 이 어트리뷰트들에 항상 문자열이 포함되도록 하고 싶습니다.

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
    }
}

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    tmp = self->last;
    Py_INCREF(value);
    self->last = value;
    Py_DECREF(tmp);
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

        .tp_itemsize = 0,
        .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
        .tp_new = Custom_new,
        .tp_init = (initproc) Custom_init,
        .tp_dealloc = (destructor) Custom_dealloc,
        .tp_members = Custom_members,
        .tp_methods = Custom_methods,
        .tp_getset = Custom_getsetters,
    };

    static PyModuleDef custommodule = {
        PyModuleDef_HEAD_INIT,
        .m_name = "custom3",
        .m_doc = "Example module that creates an extension type.",
        .m_size = -1,
    };

    PyMODINIT_FUNC
    PyInit_custom3(void)
    {
        PyObject *m;
        if (PyType_Ready(&CustomType) < 0)
            return NULL;

        m = PyModule_Create(&custommodule);
        if (m == NULL)
            return NULL;

        Py_INCREF(&CustomType);
        if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
            Py_DECREF(&CustomType);
            Py_DECREF(m);
            return NULL;
        }

        return m;
    }

```

first와 last 어트리뷰트를 더 효과적으로 제어하기 위해, 사용자 정의 게터(getter)와 세터(setter) 함수를 사용합니다. first 어트리뷰트를 가져오고 설정하는 함수는 다음과 같습니다:

```

    static PyObject *
    Custom_getfirst(CustomObject *self, void *closure)
    {
        Py_INCREF(self->first);
        return self->first;
    }

    static int
    Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
    {
        PyObject *tmp;
        if (value == NULL) {
            PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
            return -1;
        }
        if (!PyUnicode_Check(value)) {
            PyErr_SetString(PyExc_TypeError,
                            "The first attribute value must be a string");
            return -1;
        }
    }

```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```

tmp = self->first;
Py_INCREF(value);
self->first = value;
Py_DECREF(tmp);
return 0;
}
    
```

게터 (getter) 함수에는 Custom 객체와 《클로저 (closure)》 (void 포인터)가 전달됩니다. 이 경우, 클로저는 무시됩니다. (클로저는 정의 데이터가 게터 (getter)와 세터 (setter)로 전달되는 고급 사용법을 지원합니다. 예를 들어, 클로저의 데이터에 기반하여 가져오거나 설정할 어트리뷰트를 결정하는 단일 게터 (getter)와 세터 (setter) 함수 집합을 가능하게 합니다.)

세터 (setter) 함수에는 Custom 객체, 새 값 및 클로저 (closure)가 전달됩니다. 새 값은 NULL일 수 있으며, 이 경우 어트리뷰트가 삭제됩니다. 세터 (setter)에서, 우리는 어트리뷰트가 삭제되거나 새 값이 문자열이 아니면 에러를 발생시킵니다.

PyGetSetDef 구조체의 배열을 만듭니다:

```

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};
    
```

그리고 tp\_getset 슬롯에 등록합니다:

```

.tp_getset = Custom_getsetters,
    
```

PyGetSetDef 구조체의 마지막 항목은 위에서 언급한 《클로저》입니다. 이 경우, 클로저를 사용하지 않아서, NULL만 전달합니다.

또한 이러한 어트리뷰트에 대한 멤버 정의를 제거합니다:

```

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};
    
```

또한 문자열만 전달되도록<sup>3</sup> tp\_init 처리기를 갱신해야 합니다:

```

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
    }
}
    
```

(다음 페이지에 계속)

<sup>3</sup> 우리는 이제 first와 last 멤버가 문자열이라는 것을 알고 있어서, 참조 횟수를 줄이는 데 덜 주의할 수 있지만, 우리는 문자열 서브 클래스의 인스턴스를 받아들입니다. 일반 문자열을 할당 해제하는 것이 우리 객체로 다시 호출되지 않는더라도, 문자열 서브 클래스의 인스턴스를 할당 해제하는 것이 객체로 다시 호출되지 않는다고 보장할 수 없습니다.

(이전 페이지에서 계속)

```

        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

```

이러한 변경을 통해, `first`와 `last` 멤버가 절대 `NULL`이 아니라고 확신할 수 있어서, 거의 모든 경우에 `NULL` 값 검사를 제거할 수 있습니다. 이것은 대부분의 `Py_XDECREF()` 호출이 `Py_DECREF()` 호출로 변환될 수 있음을 의미합니다. 이러한 호출을 변경할 수 없는 유일한 장소는 `tp_dealloc` 구현에서인데, `tp_new`에서 이 멤버의 초기화가 실패했을 가능성이 있습니다.

또한 이전과 마찬가지로, 초기화 함수에서 모듈 초기화 함수와 모듈 이름을 바꾸고, `setup.py` 파일에 추가 정의를 추가합니다.

## 2.2.4 순환 가비지 수거 지원하기

파이썬에는 참조 횟수가 0이 아닐 때도 불필요한 객체를 식별할 수 있는 순환 가비지 수거기 (GC)가 있습니다. 이것은 객체가 순환에 참여할 때 일어날 수 있습니다. 예를 들어, 다음을 고려하십시오:

```

>>> l = []
>>> l.append(l)
>>> del l

```

이 예에서, 자신을 포함하는 리스트를 만듭니다. 삭제해도 여전히 자체 참조가 있습니다. 참조 횟수가 0으로 떨어지지 않습니다. 다행스럽게도, 파이썬의 순환 가비지 수거기는 결국 리스트가 가비지임을 확인하고 해제합니다.

`Custom` 예제의 두 번째 버전에서는, 모든 종류의 객체를 `first`나 `last` 어트리뷰트에 저장할 수 있었습니다<sup>4</sup>. 게다가 두 번째와 세 번째 버전에서는, `Custom`을 서브클래싱 할 수 있었고, 서브클래스는 임의의 어트리뷰트를 추가할 수 있습니다. 이 두 가지 이유 중 어느 것으로도, `Custom` 객체는 순환에 참여할 수 있습니다:

```

>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n

```

순환 GC가 참조 순환에 참여하는 `Custom` 인스턴스를 올바르게 감지하고 수집할 수 있도록, `Custom` 형은 두 개의 추가 슬롯을 채우고 이러한 슬롯을 활성화하는 플래그를 활성화해야 합니다:

```

#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

```

(다음 페이지에 계속)

<sup>4</sup> 또한, 어트리뷰트가 문자열 인스턴스로 제한된 경우에도, 사용자는 임의의 `str` 서브클래스를 전달할 수 있어서 여전히 참조 순환을 만들 수 있습니다.

(이전 페이지에서 계속)

```

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}

static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
}

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->first);
    self->first = value;
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    Py_CLEAR(self->last);
    self->last = value;
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_traverse = (traverseproc) Custom_traverse,
    .tp_clear = (inquiry) Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;
}

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Py_INCREF(&CustomType);
if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
    Py_DECREF(&CustomType);
    Py_DECREF(m);
    return NULL;
}

return m;
}
```

첫째, 탐색(traversal) 메서드는 순환 GC가 순환에 참여할 수 있는 서브 객체에 대해 알 수 있도록 합니다:

```
static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    int vret;
    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }
    return 0;
}
```

순환에 참여할 수 있는 각 서브 객체에 대해 탐색 메서드에 전달되는 `visit()` 함수를 호출해야 합니다. `visit()` 함수는 서브 객체와 탐색 메서드에 전달한 추가 인자 `arg` 인자를 인자로 취합니다. 0이 아닌 경우 반환해야 하는 정숫값을 반환합니다.

파이썬은 `visit` 함수 호출을 자동화하는 `Py_VISIT()` 매크로를 제공합니다. `Py_VISIT()`를 사용하면, `Custom_traverse`에서 관용구 양을 최소화할 수 있습니다:

```
static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

**참고:** `Py_VISIT()`를 사용하려면 `tp_traverse` 구현에서 인자 이름을 `visit`과 `arg`로 정확하게 지정해야 합니다.

둘째, 순환에 참여할 수 있는 서브 객체를 지우는 메서드를 제공해야 합니다:

```
static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

`Py_CLEAR()` 매크로 사용에 주목하십시오. 참조 횟수를 줄이면서 임의 형의 데이터 어트리뷰트를 지우는 권장되고 안전한 방법입니다. `NULL`로 설정하기 전에 어트리뷰트에서 `Py_XDECREF()`를 대신 호출했으면,

어트리뷰트의 파괴자가 어트리뷰트를 다시 읽는 코드(특히 참조 순환이 있으면)를 다시 호출할 가능성이 있습니다.

**참고:** 다음과 같이 작성하여 `Py_CLEAR()` 를 에뮬레이션할 수 있습니다:

```
PyObject *tmp;
tmp = self->first;
self->first = NULL;
Py_XDECREF(tmp);
```

그런데도, 어트리뷰트를 삭제할 때 항상 `Py_CLEAR()` 를 사용하기가 훨씬 쉽고 예러가 적습니다. 견고성을 희생하면서 세밀한 최적화를 시도하지 마십시오!

할당 해제기 `Custom_dealloc`은 어트리뷰트를 지울 때 임의의 코드를 호출할 수 있습니다. 이는 함수 내에서 순환 GC가 트리거 될 수 있음을 의미합니다. GC는 참조 횟수가 0이 아니라고 가정하기 때문에, 멤버를 지우기 전에 `PyObject_GC_UnTrack()` 을 호출하여 GC에서 객체를 추적 해제해야 합니다. 다음은 `PyObject_GC_UnTrack()` 과 `Custom_clear`를 사용하여 다시 구현된 할당 해제기입니다:

```
static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

마지막으로, `Py_TPFLAGS_HAVE_GC` 플래그를 클래스 플래그에 추가합니다:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

거의 다 됐습니다. 사용자 정의 `tp_alloc`이나 `tp_free` 처리기를 작성했으면, 순환 가비지 수거를 위해 이를 수정해야 합니다. 대부분의 확장은 자동으로 제공된 버전을 사용합니다.

## 2.2.5 다른 형의 서브 클래싱

기존 형에서 파생된 새 확장형을 만들 수 있습니다. 확장이 필요한 `PyTypeObject`를 쉽게 사용할 수 있어서, 내장형에서 상속하기가 가장 쉽습니다. 확장 모듈 간에 이러한 `PyTypeObject` 구조체를 공유하기 어려울 수 있습니다.

이 예에서는 내장 `list` 형을 상속하는 `SubList` 형을 만듭니다. 새로운 형은 일반 리스트와 완전히 호환되지만, 내부 카운터를 증가시키는 추가 `increment()` 메서드가 있습니다:

```
>>> import sublist
>>> s = sublist.SubList(range(3))
>>> s.extend(s)
>>> print(len(s))
6
>>> print(s.increment())
1
>>> print(s.increment())
2
```

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

typedef struct {
    PyListObject list;
    int state;
} SubListObject;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

static PyObject *
SubList_increment(SubListObject *self, PyObject *unused)
{
    self->state++;
    return PyLong_FromLong(self->state);
}

static PyMethodDef SubList_methods[] = {
    {"increment", (PyCFunction) SubList_increment, METH_NOARGS,
     PyDoc_STR("increment state counter")},
    {NULL},
};

static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}

static PyTypeObject SubListType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist.SubList",
    .tp_doc = "SubList objects",
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_init = (initproc) SubList_init,
    .tp_methods = SubList_methods,
};

static PyModuleDef sublistmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject *m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
    
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

}

보시다시피, 소스 코드는 이전 섹션의 Custom 예제와 매우 유사합니다. 우리는 그들 사이의 주요 차이점을 분석할 것입니다.

```
typedef struct {
    PyObject list;
    int state;
} SubListObject;
```

파생형 객체의 주요 차이점은 베이스형의 객체 구조체가 첫 번째 값이어야 한다는 것입니다. 베이스형은 이미 구조체의 시작 부분에 PyObject\_HEAD()를 포함합니다.

파이썬 객체가 SubList 인스턴스일 때, PyObject \* 포인터는 PyObject \*와 SubListObject \* 모두로 안전하게 캐스팅될 수 있습니다:

```
static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}
```

위에서 베이스형의 \_\_init\_\_ 메서드를 호출하는 방법을 볼 수 있습니다.

이 패턴은 사용자 정의 tp\_new와 tp\_dealloc 멤버를 갖는 형을 작성할 때 중요합니다. tp\_new 처리기는 실제로 tp\_alloc을 사용하여 객체의 메모리를 만들지 말고, 베이스 클래스가 자체 tp\_new를 호출하여 처리하도록 해야 합니다.

PyTypeObject 구조체는 형의 구상 베이스 클래스를 지정하는 tp\_base를 지원합니다. 크로스 플랫폼 컴파일러 문제로 인해, PyList\_Type에 대한 참조로 해당 필드를 직접 채울 수 없습니다; 나중에 모듈 초기화 함수에서 수행해야 합니다:

```
PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject* m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

PyType\_Ready()를 호출하기 전에, 형 구조체에 tp\_base 슬롯이 채워져 있어야 합니다. 기존 형을 파생할 때, PyType\_GenericNew()로 tp\_alloc 슬롯을 채울 필요는 없습니다 - 베이스형의 할당 함수가 상속됩니다.

그런 다음, PyType\_Ready()를 호출하고 형 객체를 모듈에 추가하는 것은 기초 Custom 예와 같습니다.

## 2.3 확장형 정의하기: 여러 가지 주제

이 섹션은 구현할 수 있는 다양한 형 메서드와 그것이 하는 일에 대해 훑어보기를 제공하기 위한 것입니다. 다음은 디버그 빌드에서만 사용되는 일부 필드가 생략된 PyObject의 정의입니다:

```
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;
    getattrofunc tp_getattr;
    setattrofunc 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 */

    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

struct PyGetSetDef *tp_getset;
struct _typeobject *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;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
destructor tp_del;

/* Type attribute cache version tag. Added in version 2.6 */
unsigned int tp_version_tag;

destructor tp_finalize;
} PyTypeObject;
    
```

이제 메서드가 아주 많습니다. 너무 걱정하지 마십시오 – 정의하려는 형이 있으면, 이 중 일부만 구현할 가능성이 매우 높습니다.

아마 지금까지 예상했듯이, 이것에 대해 살펴보고 다양한 처리기에 대한 자세한 정보를 제공할 것입니다. 필드의 순서에 영향을 미치는 많은 과거의 짐이 있어서, 구조체에 정의된 순서대로 진행하지 않을 것입니다. 필요한 필드가 포함된 예제를 찾은 다음 새 형에 맞게 값을 변경하기가 종종 가장 쉽습니다.

```

const char *tp_name; /* For printing */
    
```

형의 이름 – 이전 장에서 언급했듯이, 이것은 여러 곳에서 나타나는데, 거의 진단 목적입니다. 그러한 상황에서 도움이 될만한 것을 선택하십시오!

```

Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
    
```

이 필드는 이 형의 새 객체가 만들어질 때 할당할 메모리량을 런타임에 알려줍니다. 파이썬은 가변 길이 구조(생각하세요: 문자열, 튜플)에 대한 지원을 내장하고 있는데, 이때 `tp_itemsize` 필드가 참여합니다. 이것은 나중에 다룰 것입니다.

```

const char *tp_doc;
    
```

여기에 파이썬 스크립트가 `obj.__doc__`을 참조하여 독스트링을 꺼낼 때 반환할 문자열(또는 문자열의 주소)을 넣을 수 있습니다.

이제 기본 형 메서드에 대해 살펴보겠습니다 – 대부분의 확장형이 구현할 것들입니다.

### 2.3.1 파이널리제이션과 할당 해제

```
destructor tp_dealloc;
```

이 함수는 형의 인스턴스의 참조 횟수가 0으로 줄어들고 파이썬 인터프리터가 그것을 재활용하고자 할 때 호출됩니다. 여러분의 형에 해제할 메모리가 있거나 수행할 기타 정리 작업이 있으면, 여기에 넣을 수 있습니다. 객체 자체도 여기서 해제해야 합니다. 이 함수의 예는 다음과 같습니다:

```
static void
newdatatype_dealloc(newdatatypeobject *obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    Py_TYPE(obj)->tp_free(obj);
}
```

할당 해제 함수의 중요한 요구 사항 중 하나는 계류 중인 예외를 그대로 남겨 두어야 한다는 것입니다. 인터프리터가 파이썬 스택을 되감을 때 할당 해제기가 자주 호출되기 때문에 중요합니다; 스택이 (정상적인 반환이 아닌) 예외로 인해 되감길 때, 할당 해제기가 예외가 이미 설정되어 있음을 알 수 없도록 하는 것은 아무것도 수행되지 않습니다. 할당 해제기가 수행하는 추가 파이썬 코드가 실행될 수 있도록 하는 추가 조치는 예외가 설정되었음을 감지할 수 있습니다. 이는 인터프리터가 혼동하도록 할 수 있습니다. 이를 방지하는 올바른 방법은 안전하지 않은 조치를 수행하기 전에 계류 중인 예외를 저장하고 완료되면 복원하는 것입니다. `PyErr_Fetch()` 와 `PyErr_Restore()` 함수를 사용하여 수행할 수 있습니다:

```
static void
my_dealloc(PyObject *obj)
{
    PyObject *self = (PyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;

        /* This saves the current exception state */
        PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallObject(self->my_callback, NULL);
        if (cbresult == NULL)
            PyErr_WriteUnraisable(self->my_callback);
        else
            Py_DECREF(cbresult);

        /* This restores the saved exception state */
        PyErr_Restore(err_type, err_value, err_traceback);

        Py_DECREF(self->my_callback);
    }
    Py_TYPE(obj)->tp_free((PyObject*) self);
}
```

**참고:** 할당 해제 함수에서 안전하게 수행할 수 있는 작업에는 제한이 있습니다. 먼저, 형이 가비지 수거를 지원하면(`tp_traverse` 및/또는 `tp_clear`를 사용해서), `tp_dealloc`이 호출될 때 객체의 일부 멤버가 지워지거나 파이널라이즈 될 수 있습니다. 둘째, `tp_dealloc`에서, 객체는 불안정한 상태에 있습니다: 참조 횟수가 0입니다. (위의 예에서와 같이) 사소하지 않은 객체나 API를 호출하면 `tp_dealloc`을 다시 호출하게 되어, 이중 해제와 충돌이 발생할 수 있습니다.

파이썬 3.4부터는, `tp_dealloc`에 복잡한 파이널리제이션 코드를 넣지 말고, 대신 새로운 `tp_finalize` 형 메서드를 사용하는 것이 좋습니다.

더 보기:

PEP 442는 새로운 파이널리제이션 체계를 설명합니다.

### 2.3.2 객체 표현

파이썬에서, 객체의 텍스트 표현을 생성하는 두 가지 방법이 있습니다: `repr()` 함수와 `str()` 함수. (`print()` 함수는 단지 `str()` 을 호출합니다.) 이 처리기들은 모두 선택적입니다.

```
reprfunc tp_repr;
reprfunc tp_str;
```

`tp_repr` 처리기는 호출된 인스턴스의 표현을 포함하는 문자열 객체를 반환해야 합니다. 다음은 간단한 예입니다:

```
static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Repr-ified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}
```

`tp_repr` 처리기가 지정되지 않으면, 인터프리터는 형의 `tp_name`과 객체의 고유 식별 값을 사용하는 표현을 제공합니다.

`tp_str` 처리기는 `str()` 에 대한 것이고, 위에서 설명한 `tp_repr` 처리기와 `repr()` 간의 관계와 같은 관계입니다; 즉, 파이썬 코드가 객체의 인스턴스에서 `str()` 을 호출할 때 호출됩니다. 구현은 `tp_repr` 함수와 매우 유사하지만, 결과 문자열은 사람이 사용하기 위한 것입니다. `tp_str`을 지정하지 않으면, `tp_repr` 처리기가 대신 사용됩니다.

다음은 간단한 예입니다:

```
static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Stringified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}
```

### 2.3.3 어트리뷰트 관리

어트리뷰트를 지원할 수 있는 모든 객체에 대해, 해당 형은 어트리뷰트가 결정되는(resolved) 방법을 제어하는 함수를 제공해야 합니다. 어트리뷰트를 꺼낼 수 있는 함수와(뭔가 정의되어 있다면), 어트리뷰트를 설정하는 다른 함수(어트리뷰트 설정이 허용된다면)가 있어야 합니다. 어트리뷰트 제거는 특별한 경우이며, 처리기에 전달된 새 값이 NULL입니다.

파이썬은 두 쌍의 어트리뷰트 처리기를 지원합니다; 어트리뷰트를 지원하는 형은 한 쌍의 함수만 구현하면 됩니다. 차이점은 한 쌍은 어트리뷰트 이름을 `char*`로 취하고, 다른 쌍은 `PyObject*`를 받아들인다는 것입니다. 각 형은 구현의 편의에 더 적합한 쌍을 사용할 수 있습니다.

```
getattrfunc tp_getattr;      /* char * version */
setattrfunc tp_setattr;
/* ... */
getattrfunc tp_getattro;     /* PyObject * version */
setattrfunc tp_setattro;
```

객체의 어트리뷰트에 액세스하는 것이 항상 간단한 연산이면(짧게 설명할 것입니다), 어트리뷰트 관리 함수의 `PyObject*` 버전을 제공하는 데 사용할 수 있는 일반적인 구현이 있습니다. 파이썬 2.2부터 형별 어트리뷰트 처리기에 대한 실제 필요성은 거의 완전히 사라졌지만, 사용 가능한 새로운 일반 메커니즘을 사용하도록 갱신되지 않은 예제가 많이 있습니다.

## 범용 어트리뷰트 관리

대부분의 확장형은 간단한 어트리뷰트만 사용합니다. 그렇다면, 어트리뷰트를 간단하게 만드는 것은 무엇입니까? 충족해야 하는 몇 가지 조건만 있습니다:

1. `PyType_Ready()` 가 호출될 때 어트리뷰트의 이름을 알아야 합니다.
2. 어트리뷰트를 찾거나 설정했음을 기록하는 데 특별한 처리가 필요하지 않으며 값을 기반으로 조치를 하지 않아도 됩니다.

이 목록은 어트리뷰트 값, 값을 계산하는 시점 또는 관련 데이터가 저장되는 방법에 제한을 두지 않음에 유의하십시오.

`PyType_Ready()` 가 호출될 때, 형 객체가 참조하는 3개의 테이블을 사용하여 형 객체의 디렉터리에서 배치되는 **디스크립터**를 만듭니다. 각 디스크립터는 인스턴스 객체의 한 어트리뷰트에 대한 액세스를 제어합니다. 각 테이블은 선택적입니다; 세 개 모두가 NULL이면, 형의 인스턴스는 베이스형에서 상속된 어트리뷰트만 갖게 되며, `tp_getattro`와 `tp_setattro` 필드도 NULL로 남겨두어야 베이스형이 어트리뷰트를 처리할 수 있습니다.

테이블은 형 객체의 세 필드로 선언됩니다:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

`tp_methods`가 NULL이 아니면, `PyMethodDef` 구조체의 배열을 참조해야 합니다. 테이블의 각 항목은 다음 구조체의 인스턴스입니다:

```
typedef struct PyMethodDef {
    const char *ml_name;           /* method name */
    PyCFunction ml_meth;           /* implementation function */
    int ml_flags;                  /* flags */
    const char *ml_doc;            /* docstring */
} PyMethodDef;
```

형에서 제공되는 각 메서드에 대해 하나의 항목을 정의해야 합니다; 베이스형에서 상속된 메서드에는 항목이 필요하지 않습니다. 마지막에 하나의 추가 항목이 필요합니다; 배열의 끝을 나타내는 센티넬(sentinel)입니다. 센티넬의 `ml_name` 필드는 NULL이어야 합니다.

두 번째 테이블은 인스턴스에 저장된 데이터에 직접 매핑되는 어트리뷰트를 정의하는 데 사용됩니다. 다양한 기본 C형이 지원되며, 액세스는 읽기 전용이거나 읽고 쓰기일 수 있습니다. 테이블의 구조체는 다음과 같이 정의됩니다:

```
typedef struct PyMemberDef {
    const char *name;
    int type;
    int offset;
    int flags;
    const char *doc;
} PyMemberDef;
```

테이블의 각 항목에 대해, **디스크립터**가 구성되고 형에 추가되어 인스턴스 구조체에서 값을 추출할 수 있게 됩니다. `type` 필드는 `structmember.h` 헤더에 정의된 형 코드 중 하나를 포함해야 합니다; 이 값은 파이썬 값과 C값 간에 변환하는 방법을 결정하는 데 사용됩니다. `flags` 필드는 어트리뷰트에 액세스하는 방법을 제어하는 플래그를 저장하는 데 사용됩니다.

다음 플래그 상수는 `structmember.h`에 정의되어 있습니다; 비트별 OR를 사용하여 결합할 수 있습니다.

상수	의미
READONLY	쓸 수 없습니다.
READ_RESTRICTED	제한된 모드에서는 읽을 수 없습니다.
WRITE_RESTRICTED	제한된 모드에서는 쓸 수 없습니다.
RESTRICTED	제한된 모드에서는 읽거나 쓸 수 없습니다.

tp\_members 테이블을 사용하여 실행 시간에 사용되는 디스크립터를 구축하는 것의 흥미로운 이점은 이 방법으로 정의된 모든 어트리뷰트가 단순히 테이블에 텍스트를 제공하는 것으로 연관된 독스트링을 가질 수 있다는 것입니다. 응용 프로그램은 내부 검사(introspection) API를 사용하여 클래스 객체에서 디스크립터를 꺼내고, 그것의 \_\_doc\_\_ 어트리뷰트를 사용하여 독스트링을 얻을 수 있습니다.

tp\_methods 테이블과 마찬가지로, name 값이 NULL인 센티넬 항목이 필요합니다.

## 형별 어트리뷰트 관리

간단히 하기 위해, char\* 버전 만 여기에서 예시합니다; name 매개 변수의 형이 인터페이스의 char\*와 PyObject\* 버전 간의 유일한 차이점입니다. 이 예제는 위의 범용 예제와 효과적으로 같은 것을 수행하지만, 파이썬 2.2에 추가된 범용 지원은 사용하지 않습니다. 처리기 함수가 호출되는 방식을 설명하므로, 기능을 확장해야 한다면, 무엇을 해야 할지 이해할 수 있을 겁니다.

tp\_getattr 처리기는 객체에 어트리뷰트 조회가 필요할 때 호출됩니다. 클래스의 \_\_getattr\_\_() 메서드가 호출되는 것과 같은 상황에서 호출됩니다.

예는 다음과 같습니다:

```
static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    if (strcmp(name, "data") == 0)
    {
        return PyLong_FromLong(obj->data);
    }

    PyErr_Format(PyExc_AttributeError,
                 "'%.50s' object has no attribute '%.400s'",
                 tp->tp_name, name);
    return NULL;
}
```

tp\_setattr 처리기는 클래스 인스턴스의 \_\_setattr\_\_() 이나 \_\_delattr\_\_() 메서드가 호출될 때 호출됩니다. 어트리뷰트를 삭제해야 하면, 세 번째 매개 변수는 NULL이 됩니다. 다음은 단순히 예외를 발생시키는 예입니다; 이것이 정말로 여러분이 원하는 전부라면, tp\_setattr 처리기는 NULL로 설정되어야 합니다.

```
static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *v)
{
    PyErr_Format(PyExc_RuntimeError, "Read-only attribute: %s", name);
    return -1;
}
```

## 2.3.4 객체 비교

```
richcmpfunc tp_richcompare;
```

tp\_richcompare 처리기는 비교가 필요할 때 호출됩니다. \_\_lt\_\_() 와 같은 풍부한 비교 메서드에 해당하며, PyObject\_RichCompare() 와 PyObject\_RichCompareBool() 에 의해서도 호출됩니다.

이 함수는 두 개의 파이썬 객체와 연산자를 인자로 사용하여 호출됩니다, 여기서 연산자는 Py\_EQ, Py\_NE, Py\_LE, Py\_GT, Py\_LT 또는 Py\_GE 중 하나입니다. 지정된 연산자로 두 객체를 비교하고 비교에 성공하면 Py\_True나 Py\_False를, 비교가 구현되지 않았으며 다른 객체의 비교 메서드를 시도해야 한다는 것을 나타내려면 Py\_NotImplemented를, 예외가 설정되면 NULL을 반환해야 합니다.

내부 포인터의 크기가 같으면 같다고 간주하는 데이터형에 대한 샘플 구현은 다음과 같습니다:



```
static PyObject *
newdatatype_richcmp(PyObject *obj1, PyObject *obj2, int op)
{
    PyObject *result;
    int c, size1, size2;

    /* code to make sure that both arguments are of type
       newdatatype omitted */

    size1 = obj1->obj_UnderlyingDatatypePtr->size;
    size2 = obj2->obj_UnderlyingDatatypePtr->size;

    switch (op) {
        case Py_LT: c = size1 < size2; break;
        case Py_LE: c = size1 <= size2; break;
        case Py_EQ: c = size1 == size2; break;
        case Py_NE: c = size1 != size2; break;
        case Py_GT: c = size1 > size2; break;
        case Py_GE: c = size1 >= size2; break;
    }
    result = c ? Py_True : Py_False;
    Py_INCREF(result);
    return result;
}
```

## 2.3.5 추상 프로토콜 지원

파이썬은 다양한 추상 <프로토콜>을 지원합니다; 이러한 인터페이스를 사용하기 위해 제공되는 구체적인 인터페이스는 `abstract`에 설명되어 있습니다.

이러한 추상 인터페이스 중 다수는 파이썬 구현 개발 초기에 정의되었습니다. 특히, 숫자, 매핑 및 시퀀스 프로토콜은 처음부터 파이썬의 일부였습니다. 다른 프로토콜은 시간이 지남에 따라 추가되었습니다. 형 구현의 여러 처리기 루틴에 의존하는 프로토콜의 경우, 이전 프로토콜은 형 객체가 참조하는 선택적 처리기 블록으로 정의되었습니다. 최신 프로토콜의 경우 메인 형 객체에 추가 슬롯이 있으며, 슬롯이 존재하고 인터프리터가 확인해야 함을 나타내는 플래그 비트가 설정됩니다. (플래그 비트는 슬롯 값이 NULL이 아님을 나타내지 않습니다. 플래그는 슬롯의 존재를 나타내도록 설정될 수 있지만, 슬롯은 여전히 채워지지 않을 수 있습니다.)

```
PyNumberMethods    *tp_as_number;
PySequenceMethods  *tp_as_sequence;
PyMappingMethods    *tp_as_mapping;
```

여러분의 객체가 숫자, 시퀀스 또는 매핑 객체처럼 작동하도록 하려면, C형 `PyNumberMethods`, `PySequenceMethods` 또는 `PyMappingMethods`를 각각 구현하는 구조체의 주소를 배치합니다. 이 구조체를 적절한 값으로 채우는 것은 여러분의 책임입니다. 파이썬 소스 배포의 `Objects` 디렉터리에서 이들 각각의 사용 예를 찾을 수 있습니다.

```
hashfunc tp_hash;
```

여러분이 제공하기로 선택했다면, 이 함수는 데이터형의 인스턴스에 대한 해시 숫자를 반환해야 합니다. 다음은 간단한 예입니다:

```
static Py_hash_t
newdatatype_hash(newdatatypeobject *obj)
{
    Py_hash_t result;
    result = obj->some_size + 32767 * obj->some_number;
    if (result == -1)
        result = -2;
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```
return result;
}
```

Py\_hash\_t는 플랫폼에 따라 변하는 너비의 부호 있는 정수 형입니다. tp\_hash에서 -1을 반환하면 에러를 표시해서, 위와 같이 해시 계산에 성공했을 때 반환하지 않도록 주의해야 합니다.

```
ternaryfunc tp_call;
```

이 함수는 데이터형의 인스턴스가 《호출》될 때 호출됩니다, 예를 들어, obj1이 데이터형의 인스턴스이고 파이썬 스크립트에 obj1('hello')가 포함되어 있으면 tp\_call 처리기가 호출됩니다.

이 함수는 세 개의 인자를 취합니다:

1. *self*는 호출의 대상인 데이터형의 인스턴스입니다. 호출이 obj1('hello')이면, *self*는 obj1입니다.
2. *args*는 호출에 대한 인자를 포함하는 튜플입니다. PyArg\_ParseTuple()을 사용하여 인자를 추출할 수 있습니다.
3. *kws*는 전달된 키워드 인자의 딕셔너리입니다. 이것이 NULL이 아니고 키워드 인자를 지원하면 PyArg\_ParseTupleAndKeywords()를 사용하여 인자를 추출하십시오. 키워드 인자를 지원하지 않고 이것이 NULL이 아니면, 키워드 인자가 지원되지 않는다는 메시지와 함께 TypeError를 발생시키십시오.

장난감 tp\_call 구현은 다음과 같습니다:

```
static PyObject *
newdatatype_call(newdatatypeobject *self, PyObject *args, PyObject *kws)
{
    PyObject *result;
    const char *arg1;
    const char *arg2;
    const char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
        return NULL;
    }
    result = PyUnicode_FromFormat(
        "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3: [%s]\n",
        obj->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
    return result;
}
```

```
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

이 함수는 이터레이터 프로토콜 지원을 제공합니다. 두 처리기 모두 정확히 하나의 매개 변수, 호출되는 인스턴스를 취하고 새 참조를 반환합니다. 에러가 발생하면, 예외를 설정하고 NULL을 반환해야 합니다. tp\_iter는 파이썬 \_\_iter\_\_() 메서드에 해당하고, tp\_iternext는 파이썬 \_\_next\_\_() 메서드에 해당합니다.

모든 이터러블 객체는 이터레이터 객체를 반환해야 하는 tp\_iter 처리기를 구현해야 합니다. 다음은 파이썬 클래스에도 적용되는 공통 지침입니다:

- 여러 개의 독립 이터레이터를 지원할 수 있는 컬렉션(가령 리스트와 튜플)의 경우, tp\_iter를 호출할 때마다 새 이터레이터가 만들어지고 반환되어야 합니다.
- 한 번만 이터레이트 될 수 있는(보통 파일 객체처럼 이터레이션의 부작용으로 인해) 객체는 스스로에 대한 새로운 참조를 반환하여 tp\_iter를 구현할 수 있습니다 – 따라서 tp\_iternext 처리기도 구현해야 합니다.

모든 **이터레이터** 객체는 `tp_iter`와 `tp_iternext`를 모두 구현해야 합니다. 이터레이터의 `tp_iter` 처리기는 이터레이터에 대한 새로운 참조를 반환해야 합니다. `tp_iternext` 처리기는 이터레이션의 다음 객체(있다면)에 대한 새 참조를 반환해야 합니다. 이터레이션이 끝에 도달하면, `tp_iternext`는 예외를 설정하지 않고 `NULL`을 반환하거나, `NULL`을 반환하는 것에 더해 `StopIteration`을 설정할 수 있습니다; 예외를 피하면 성능이 약간 향상될 수 있습니다. 실제 예러가 발생하면, `tp_iternext`는 항상 예외를 설정하고, `NULL`을 반환해야 합니다.

## 2.3.6 약한 참조 지원

파이썬의 약한 참조 구현의 목표 중 하나는 성능에 중요한 객체(가령 숫자)에 대한 부하를 발생시키지 않고 모든 형이 약한 참조 메커니즘에 참여할 수 있도록 하는 것입니다.

더 보기:

`weakref` 모듈에 대한 설명서.

객체가 약하게 참조될 수 있으려면, 확장형이 두 가지 작업을 수행해야 합니다:

1. 약한 참조 메커니즘 전용 C 객체 구조체에 `PyObject*` 필드를 포함하십시오. 객체의 생성자는 이것을 `NULL`로 남겨 두어야 합니다(기본 `tp_alloc`을 사용할 때는 자동입니다).
2. 인터프리터가 해당 필드에 액세스하고 수정하는 방법을 알 수 있도록, `tp_weaklistoffset` 형 멤버를 C 객체 구조체에서 위에서 언급한 필드의 오프셋으로 설정하십시오.

구체적으로, 다음은 필수 필드로 사소한 객체 구조체를 확장하는 방법입니다:

```
typedef struct {
    PyObject_HEAD
    PyObject *weakreflist; /* List of weak references */
} TrivialObject;
```

그리고 정적으로 선언된 형 객체의 해당 멤버:

```
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... other members omitted for brevity ... */
    .tp_weaklistoffset = offsetof(TrivialObject, weakreflist),
};
```

유일한 추가 사항은 필드가 `NULL`이 아니면 `tp_dealloc`이 (`PyObject_ClearWeakRefs()`를 호출하여) 모든 약한 참조를 지울 필요가 있다는 것입니다:

```
static void
Trivial_dealloc(TrivialObject *self)
{
    /* Clear weakrefs first before calling any destructors */
    if (self->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) self);
    /* ... remainder of destruction code omitted for brevity ... */
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

### 2.3.7 추가 제안

새 데이터형에 특정 메서드를 구현하는 방법을 배우려면, *CPython* 소스 코드를 구하십시오. Objects 디렉터리로 이동한 다음, C 소스 파일에서 tp\_에 원하는 기능을 더한 것(예를 들어, tp\_richcompare)을 검색하십시오. 구현하려는 함수의 예를 찾을 수 있을 겁니다.

객체가 구현 중인 형의 구상 인스턴스인지 확인해야 하면, PyObject\_TypeCheck() 함수를 사용하십시오. 사용 예는 다음과 같습니다:

```
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

더 보기:

CPython 소스 릴리스를 다운로드하십시오. <https://www.python.org/downloads/source/>

GitHub의 CPython 프로젝트, CPython 소스 코드가 개발되는 곳. <https://github.com/python/cpython>

## 2.4 C와 C++ 확장 빌드하기

CPython의 C 확장은 초기화 함수를 내보내는 공유 라이브러리입니다(예를 들어, 리눅스는 .so, 윈도우는 .pyd).

임포트 할 수 있으려면, 공유 라이브러리가 PYTHONPATH에 있어야 하며, 모듈 이름을 따라 적절한 확장자를 붙여서 이름 지어야 합니다. distutils를 사용하면, 올바른 파일 이름이 자동으로 생성됩니다.

초기화 함수는 다음과 같은 서명을 갖습니다:

PyObject\* PyInit\_modulename(void)

완전히 초기화된 모듈이나 PyModuleDef 인스턴스를 반환합니다. 자세한 내용은 initializing-modules을 참조하십시오.

ASCII로만 이루어진 이름을 가진 모듈의 경우, 함수의 이름을 PyInit\_<modulename>이어야 합니다. 여기서 <modulename>을 모듈의 이름으로 치환합니다. multi-phase-initialization를 사용할 때 ASCII가 아닌 모듈 이름이 허용됩니다. 이 경우, 초기화 함수 이름은 PyInitU\_<modulename>이며 <modulename>은 파이썬의 punycode 인코딩으로 인코딩되고 하이픈을 밑줄로 대체합니다. 파이썬에서:

```
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
```

여러 초기화 함수를 정의하여 단일 공유 라이브러리에서 여러 모듈을 내보낼 수 있습니다. 그러나, 이들을 임포트 하려면 심볼릭 링크나 사용자 정의 임пор터를 사용해야 합니다. 기본적으로 파일 이름에 해당하는 함수만 발견되기 때문입니다. 자세한 내용은 PEP 489의 《한 라이브러리에 여러 모듈》절을 참조하십시오.

## 2.4.1 distutils로 C와 C++ 확장 빌드하기

확장 모듈은 파이썬에 포함된 distutils를 사용하여 빌드할 수 있습니다. distutils가 바이너리 패키지의 생성을 지원하기 때문에, 사용자는 확장을 설치하기 위해 꼭 컴파일러와 distutils가 필요하지는 않습니다.

distutils 패키지에는 드라이버 스크립트인 setup.py가 들어 있습니다. 이것은 평범한 파이썬 파일인데, 대부분 간단한 경우에 이런 식입니다:

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       ext_modules = [module1])
```

이 setup.py와 파일 demo.c로 다음을 실행하면

```
python setup.py build
```

demo.c를 컴파일하고, build 디렉터리에 demo라는 확장 모듈을 생성합니다. 시스템에 따라, 모듈 파일은 build/lib.system 하위 디렉터리에 들어가고, demo.so나 demo.pyd와 같은 이름을 가질 수 있습니다.

setup.py에서, 모든 실행은 setup 함수를 호출하여 수행됩니다. 이것은 다양한 키워드 인자를 받아들입니다. 위의 예에서는 일부만 사용합니다. 구체적으로, 이 예는 패키지를 빌드하기 위한 메타 정보를 지정하고 패키지의 내용을 지정합니다. 일반적으로, 패키지는 파이썬 소스 모듈, 문서, 서브 패키지 등과 같은 추가 모듈이 포함됩니다. distutils의 기능에 대한 자세한 내용은 distutils-index의 distutils 설명서를 참조하십시오; 이 절에서는 확장 모듈을 빌드하는 것만 설명합니다.

드라이버 스크립트를 더 잘 구조화하기 위해, setup()에 대한 인자를 미리 계산하는 것이 일반적입니다. 위의 예에서, setup()에 대한 ext\_modules 인자는 확장 모듈의 리스트며, 각 모듈은 Extension의 인스턴스입니다. 이 예에서, 인스턴스는 단일 소스 파일 demo.c를 컴파일하여 빌드하는 demo라는 확장을 정의합니다.

많은 경우, 확장을 빌드하는 것은 더 복잡합니다. 왜냐하면, 추가적인 전처리기 정의와 라이브러리가 필요할 수 있기 때문입니다. 이는 아래에서 예시합니다.

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    define_macros = [('MAJOR_VERSION', '1'),
                                    ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       author = 'Martin v. Loewis',
       author_email = 'martin@v.loewis.de',
       url = 'https://docs.python.org/extending/building',
       long_description = '''
This is really just a demo package.
''',
       ext_modules = [module1])
```

이 예에서, setup()는 추가 메타 정보로 호출되며, 배포 패키지를 빌드해야 할 때 권장됩니다. 확장 자체에 대해서는, 전처리기 정의, 인클루드 디렉터리, 라이브러리 디렉터리 및 라이브러리를 지정합니다.

컴파일러에 따라, distutils는 이 정보를 다양한 방법으로 컴파일러에 전달합니다. 예를 들어, 유닉스에서는 다음과 같은 컴파일 명령으로 이어질 수 있습니다

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=1 -DMINOR_
↪VERSION=0 -I/usr/local/include -I/usr/local/include/python2.2 -c demo.c -o build/
↪temp.linux-i686-2.2/demo.o

gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl83 -o build/lib.
↪linux-i686-2.2/demo.so
```

이 줄은 예시 목적일 뿐입니다; distutils 사용자는 distutils가 올바르게 호출한다고 믿어야 합니다.

## 2.4.2 확장 모듈 배포하기

확장이 성공적으로 빌드되면, 이를 사용하는 세 가지 방법이 있습니다.

최종 사용자는 보통 모듈을 설치하고 싶을 것이고, 다음을 실행합니다

```
python setup.py install
```

모듈 관리자는 소스 패키지를 생성해야 합니다; 그러려면, 이렇게 실행합니다

```
python setup.py sdist
```

때에 따라, 추가 파일을 소스 배포에 포함해야 합니다; 이 작업은 MANIFEST.in 파일을 통해 수행됩니다; 자세한 내용은 manifest를 참조하십시오.

소스 배포가 성공적으로 빌드되면, 관리자는 바이너리 배포도 만들 수 있습니다. 플랫폼에 따라, 이를 위해 다음 명령 중 하나를 사용할 수 있습니다.

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```

## 2.5 윈도우에서 C와 C++ 확장 빌드하기

이 장에서는 Microsoft Visual C++를 사용하여 파이썬 용 윈도우 확장 모듈을 만드는 방법을 간략히 설명하고, 이 확장 모듈의 작동 방식에 대한 보다 자세한 배경 정보를 제공합니다. 설명 자료는 파이썬 확장을 빌드하는 법을 배우는 윈도우 프로그래머와 유닉스와 윈도우 모두에서 성공적으로 빌드 할 수 있는 소프트웨어 제작에 관심이 있는 유닉스 프로그래머 모두에게 유용합니다.

모듈 저자는 확장 모듈을 빌드하는데 이 섹션에서 설명하는 것 대신 distutils 접근 방식을 사용하는 것이 좋습니다. 파이썬을 빌드하는 데 사용된 C 컴파일러가 여전히 필요합니다; 보통 Microsoft Visual C++입니다.

**참고:** 이 장에서는 인코딩된 파이썬 버전 번호를 포함하는 여러 파일 이름을 언급합니다. 이 파일 이름은 XY로 나타낸 버전 번호로 표시됩니다; 실제로, 'X'는 주(major) 버전 번호이고 'Y'는 여러분이 작업 중인 파이썬 배포의 부(minor) 버전 번호입니다. 예를 들어, 파이썬 2.2.1을 사용하면, XY는 실제로는 22가 됩니다.

## 2.5.1 요리책 접근법

유닉스에서처럼, 윈도우에서 확장 모듈을 빌드하는 두 가지 접근법이 있습니다: `distutils` 패키지를 사용하여 빌드 프로세스를 제어하거나 수동으로 작업합니다. `distutils` 접근법은 대부분 확장에서 잘 작동합니다; `distutils`를 사용하여 확장 모듈을 빌드하고 패키징하는 방법에 대한 설명은 `distutils-index`에 있습니다. 수동으로 작업할 수밖에 없다면, `winsound` 표준 라이브러리 모듈의 프로젝트 파일을 연구하는 것이 도움이 될 겁니다.

## 2.5.2 유닉스와 윈도우의 차이점

유닉스와 윈도우는 코드의 실행시간 로딩에 완전히 다른 패러다임을 사용합니다. 동적으로 로드 할 수 있는 모듈을 빌드하려고 시도하기 전에, 시스템 작동 방식을 알고 있어야 합니다.

유닉스에서, 공유 오브젝트 (.so) 파일은 프로그램에서 사용할 코드와 프로그램에서 찾을 것으로 예상되는 함수와 데이터의 이름을 포함합니다. 파일이 프로그램에 결합할 때, 파일의 코드에 있는 함수와 데이터의 모든 참조가 함수와 데이터가 메모리에 놓이게 되는 프로그램에서의 실제 위치를 가리키도록 변경됩니다. 이것은 기본적으로 링크 작업입니다.

윈도우에서, 동적 연결 라이브러리 (.dll) 파일에는 매달린 (dangling) 참조가 없습니다. 대신, 함수나 데이터에 대한 액세스는 참조 테이블 (lookup table)을 통해 이루어집니다. 따라서 DLL 코드는 프로그램의 메모리를 참조하도록 실행 시간에 수정될 필요가 없습니다; 대신, 코드는 이미 DLL의 참조 테이블을 사용하고 있고, 실행 시간에 참조 테이블이 함수와 데이터를 가리키도록 수정됩니다.

유닉스에는, 한가지 유형의 라이브러리 파일 (.a) 만 있는데, 여러 오브젝트 파일 (.o)의 코드가 포함됩니다. 공유 오브젝트 파일 (.so)을 만들기 위한 링크 단계에서, 링커는 식별자가 정의된 위치를 알 수 없음을 발견할 수 있습니다. 링커는 라이브러리의 오브젝트 파일에서 그것들을 찾습니다. 발견하면, 그 오브젝트 파일의 모든 코드를 포함합니다.

윈도우에는, 두 가지 유형의 라이브러리, 정적 라이브러리와 임포트 라이브러리가 있습니다 (둘 다 .lib라고 합니다). 정적 라이브러리는 유닉스 .a 파일과 같습니다; 필요할 때 포함될 코드가 들어 있습니다. 임포트 라이브러리는 기본적으로 특정 식별자가 합법적이고 DLL이 로드될 때 프로그램에 존재하게 된다고 링커를 안심시키기 위해서만 사용됩니다. 따라서 링커는 임포트 라이브러리의 정보를 사용하여 DLL에 포함되지 않은 식별자를 사용하는 참조 테이블을 작성합니다. 응용 프로그램이나 DLL이 링크될 때, 임포트 라이브러리가 만들어질 수 있습니다. 이것은 응용 프로그램이나 DLL의 심볼을 사용하는, 이후의 모든 DLL에 사용해야 합니다.

다른 코드 블록 A를 공유해야 하는, 두 개의 동적 로드 모듈 B와 C를 빌드한다고 가정합니다. 유닉스에서는, B.so와 C.so에 대해 링커로 A.a를 전달하지 않습니다; 전달하면 B와 C가 각각 자신의 복사본을 갖게 되어 두 번 포함하게 됩니다. 윈도우에서는, A.dll를 빌드하면 A.lib도 빌드됩니다. 여러분은 B와 C에 대해 링커로 A.lib를 전달 합니다. A.lib는 코드를 포함하지 않습니다; 실행 시간에 A의 코드에 액세스하는 데 사용될 정보만 포함합니다.

윈도우에서, 임포트 라이브러리를 사용하는 것은 `import spam`을 사용하는 것과 비슷합니다; 이것은 스팸의 이름에 액세스할 수 있도록 하지만, 별도의 복사본을 만들지는 않습니다. 유닉스에서, 라이브러리와 링크하는 것은 `from spam import *`와 더 비슷합니다; 별도의 복사본을 만듭니다.

## 2.5.3 DLL을 실제로 사용하기

윈도우 파이썬은 Microsoft Visual C++로 빌드되었습니다; 다른 컴파일러를 사용하는 것은 동작할 수도 있고 그렇지 않을 수도 있습니다 (볼랜드는 되는 것 같지만). 이 섹션의 나머지 부분은 MSVC++에만 해당합니다.

윈도우에서 DLL을 만들 때, `pythonXY.lib`를 링커에 전달해야 합니다. 두 개의 DLL, `spam`과 (`spam`에 있는 C 함수를 사용하는) `ni`를 빌드하려면, 다음 명령을 사용할 수 있습니다:

```
cl /LD /I/python/include spam.c ../libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.lib
```

첫 번째 명령은 세 개의 파일을 만들었습니다: `spam.obj`, `spam.dll` 및 `spam.lib`. `Spam.dll`은 파이썬 함수(가령 `PyArg_ParseTuple()`)를 포함하지 않지만, `pythonXY.lib` 덕분에 파이썬 코드를 찾는 방법을 알고 있습니다.

두 번째 명령은 `ni.dll`(그리고 `.obj`와 `.lib`)을 만들었습니다. `spam`과 파이썬 실행 파일에서 필요한 함수를 찾는 방법을 알고 있습니다.

모든 식별자를 참조 테이블로 보내지는 않습니다. 다른 모듈(파이썬 포함)이 식별자를 볼 수 있게 하려면, `void _declspec(dllexport) initspam(void)` 나 `PyObject _declspec(dllexport) *NiGetSpamData(void)` 처럼 `_declspec(dllexport)` 라고 선언해야 합니다.

Developer Studio는 실제로 필요하지 않은 많은 임포트 라이브러리를 던져넣어서 실행 파일에 약 100K를 추가합니다. 이것들을 제거하려면, 프로젝트 설정 대화 상자를 통해 *ignore default libraries*를 지정하십시오. 올바른 `msvcrtxx.lib`를 라이브러리 목록에 추가하십시오.





---

## 더 큰 응용 프로그램에 CPython 런타임을 내장하기

---

때로는, 파이썬 인터프리터를 메인 응용 프로그램으로 사용하고 그 안에서 실행되는 확장을 만드는 대신, CPython 런타임을 더 큰 응용 프로그램에 내장하는 것이 바람직합니다. 이 절에서는 이를 성공적으로 수행하는 데 관련된 몇 가지 세부 사항에 관해 설명합니다.

### 3.1 다른 응용 프로그램에 파이썬 내장하기

이전 장에서는 파이썬을 확장하는 방법, 즉 C 함수의 라이브러리를 파이썬에 연결하여 파이썬의 기능을 확장하는 방법에 관해 설명했습니다. 다른 방법도 가능합니다: 파이썬을 내장시켜 C/C++ 응용 프로그램을 풍부하게 만들 수 있습니다. 내장은 C 나 C++ 가 아닌 파이썬으로 응용 프로그램의 일부 기능을 구현하는 능력을 응용 프로그램에 제공합니다. 이것은 여러 목적으로 사용될 수 있습니다; 한 가지 예는 사용자가 파이썬으로 스크립트를 작성하여 응용 프로그램을 필요에 맞게 조정할 수 있게 하는 것입니다. 일부 기능을 파이썬으로 작성하기가 더 쉽다면 직접 사용할 수도 있습니다.

파이썬을 내장하는 것은 파이썬을 확장하는 것과 유사하지만, 아주 같지는 않습니다. 차이점은, 파이썬을 확장할 때 응용 프로그램의 주 프로그램은 여전히 파이썬 인터프리터입니다. 반면에 파이썬을 내장하면 주 프로그램은 파이썬과 아무 관련이 없습니다 — 대신 응용 프로그램 일부에서 간혹 파이썬 코드를 실행하기 위해 파이썬 인터프리터를 호출합니다.

그래서 파이썬을 내장한다면, 여러분은 자신의 메인 프로그램을 제공하게 됩니다. 이 메인 프로그램이 해야 할 일 중 하나는 파이썬 인터프리터를 초기화하는 것입니다. 최소한, `Py_Initialize()` 함수를 호출해야 합니다. 파이썬에 명령 줄 인자를 전달하는 선택적 호출이 있습니다. 그런 다음 나중에 응용 프로그램의 어느 부분에서나 인터프리터를 호출할 수 있습니다.

인터프리터를 호출하는 방법에는 여러 가지가 있습니다: 파이썬 문장을 포함하는 문자열을 `PyRun_SimpleString()` 에 전달하거나, `stdio` 파일 포인터와 파일명(여러 메시지에서의 식별만을 위해)을 `PyRun_SimpleFile()` 에 전달할 수 있습니다. 또한, 이전 장에서 설명한 저수준의 연산을 호출하여 파이썬 객체를 만들고 사용할 수 있습니다.

더 보기:

**c-api-index** 파이썬의 C 인터페이스에 대한 자세한 내용은 이 매뉴얼에 있습니다. 필요한 정보가 많이 있습니다.

### 3.1.1 매우 고수준의 내장

파이썬을 내장하는 가장 간단한 형태는 매우 고수준의 인터페이스를 사용하는 것입니다. 이 인터페이스는 응용 프로그램과 직접 상호 작용할 필요 없이 파이썬 스크립트를 실행하기 위한 것입니다. 이것은 예를 들어 파일에 대해 어떤 연산을 수행하는 데 사용될 수 있습니다.

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

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program); /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                      "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

`Py_SetProgramName()` 함수는 파이썬 런타임 라이브러리에 대한 경로를 인터프리터에게 알리기 위해 `Py_Initialize()` 보다 먼저 호출되어야 합니다. 다음으로, 파이썬 인터프리터는 `Py_Initialize()` 로 초기화되고, 날짜와 시간을 인쇄하는 하드 코딩된 파이썬 스크립트가 실행됩니다. 그런 다음, `Py_FinalizeEx()` 호출이 인터프리터를 종료하고 프로그램이 끝납니다. 실제 프로그램에서는 파이썬 스크립트를 다른 소스(아마도 텍스트 편집기 루틴, 파일 또는 데이터베이스)에서 가져올 수 있습니다. 파일에서 파이썬 코드를 얻는 것은 `PyRun_SimpleFile()` 함수를 사용하면 더 잘할 수 있는데, 메모리 공간을 할당하고 파일 내용을 로드하는 번거로움을 덜어줍니다.

### 3.1.2 매우 고수준 내장을 넘어서: 개요

고수준 인터페이스는 응용 프로그램에서 임의의 파이썬 코드를 실행할 수 있는 능력을 제공하지만, 최소한 데이터 값을 교환하는 것이 꽤 번거롭습니다. 그러길 원한다면 저수준의 호출을 사용해야 합니다. 더 많은 C 코드를 작성해야 하는 대신, 거의 모든 것을 달성할 수 있습니다.

파이썬을 확장하는 것과 파이썬을 내장하는 것은 다른 의도에도 불구하고 꽤 똑같은 활동이라는 점에 유의해야 합니다. 이전 장에서 논의된 대부분 주제는 여전히 유효합니다. 이것을 보시려면, 파이썬에서 C로의 확장 코드가 실제로 하는 일을 생각해봅시다:

1. 데이터값을 파이썬에서 C로 변환하고,
2. 변환된 값을 사용하여 C 루틴으로 함수 호출을 수행하고,
3. 그 호출에서 얻은 데이터값을 C에서 파이썬으로 변환합니다.

파이썬을 내장할 때, 인터페이스 코드는 다음을 수행합니다:

1. 데이터값을 C에서 파이썬으로 변환하고,
2. 변환된 값을 사용하여 파이썬 인터페이스 루틴으로 함수 호출을 수행하고,
3. 그 호출에서 얻은 데이터 값을 파이썬에서 C로 변환합니다.

보시다시피, 데이터 변환 단계가 언어 간 전송의 다른 방향을 수용하기 위해 단순히 교환됩니다. 유일한 차이점은 두 데이터 변환 간에 호출하는 루틴입니다. 확장할 때는 C 루틴을 호출하고, 내장할 때는 파이썬 루틴을 호출합니다.

이 장에서는 파이썬에서 C로 데이터를 변환하는 방법과 그 반대로 데이터를 변환하는 방법에 관해서는 설명하지 않습니다. 또한, 참조의 올바른 사용과 에러를 다루는 것을 이해하고 있다고 가정합니다. 이러한 측면은 인터프리터를 확장하는 것과 다르지 않으므로, 이전 장에서 필요한 정보를 참조할 수 있습니다.

### 3.1.3 순수한 내장

첫 번째 프로그램은 파이썬 스크립트에 있는 함수를 실행하는 것을 목표로 합니다. 매우 고수준의 인터페이스에 관한 절에서와같이, 파이썬 인터프리터는 애플리케이션과 직접 상호 작용하지 않습니다(하지만 다음 절에서 바뀔 것입니다).

파이썬 스크립트에서 정의된 함수를 실행하는 코드는 다음과 같습니다:

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

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyUnicode_DecodeFSDefault(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n", PyLong_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
            }
        }
    }
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

        fprintf(stderr, "Call failed\n");
        return 1;
    }
}
else {
    if (PyErr_Occurred())
        PyErr_Print();
    fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
}
Py_XDECREF(pFunc);
Py_DECREF(pModule);
}
else {
    PyErr_Print();
    fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
    return 1;
}
if (Py_FinalizeEx() < 0) {
    return 120;
}
return 0;
}

```

이 코드는 argv[1]를 사용하여 파이썬 스크립트를 로드하고, argv[2]에서 명명된 함수를 호출합니다. 정수 인자는 argv 배열의 남은 값들입니다. 이 프로그램을 [컴파일하고 링크하면](#) (완성된 실행 파일을 **call**이라고 부릅니다), 다음과 같은 파이썬 스크립트를 실행하는 데 사용합니다:

```

def multiply(a,b):
    print("Will compute", a, "times", b)
    c = 0
    for i in range(0, a):
        c = c + b
    return c

```

그러면 결과는 다음과 같아야 합니다:

```

$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6

```

프로그램이 기능보다 상당히 큰 편이지만, 대부분 코드는 파이썬과 C 사이의 데이터 변환과 에러 보고를 위한 것입니다. 파이썬 내장과 관련된 흥미로운 부분은 다음처럼 시작합니다

```

Py_Initialize();
pName = PyUnicode_DecodeFSDefault(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);

```

인터프리터를 초기화한 후, 스크립트는 PyImport\_Import()를 사용하여 로드됩니다. 이 루틴은 인자로 파이썬 문자열을 요구하는데, PyUnicode\_FromString() 데이터 변환 루틴을 사용하여 구성됩니다.

```

pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */

if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);

```

일단 스크립트가 로드되면, 우리가 찾고 있는 이름이 PyObject\_GetAttrString()를 사용하여 검색됩니다. 이름이 존재하고, 반환된 객체가 콜러블이면, 그것이 함수라고 안전하게 가정할 수 있습니다. 그런

다음 프로그램은 인자의 튜플을 일반적인 방법으로 구성하여 진행합니다. 그런 다음 파이썬 함수 호출은 이렇게 이루어집니다:

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

함수가 반환되면, pValue는 NULL이거나 함수의 반환 값에 대한 참조를 포함합니다. 값을 검토한 후 참조를 해제해야 합니다.

### 3.1.4 내장된 파이썬을 확장하기

지금까지 내장된 파이썬 인터프리터는 애플리케이션 자체의 기능에 액세스할 수 없었습니다. 파이썬 API는 내장된 인터프리터를 확장함으로써 이것을 허용합니다. 즉, 내장된 인터프리터는 응용 프로그램에서 제공하는 루틴으로 확장됩니다. 복잡하게 들리지만, 그렇게 나쁘지는 않습니다. 잠시 응용 프로그램이 파이썬 인터프리터를 시작한다는 것을 잊어버리십시오. 대신, 응용 프로그램을 서브 루틴의 집합으로 간주하고, 일반 파이썬 확장을 작성하는 것처럼 파이썬에서 해당 루틴에 액세스할 수 있도록 연결 코드를 작성하십시오. 예를 들면:

```
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return PyLong_FromLong(numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};

static PyModuleDef EmbModule = {
    PyModuleDef_HEAD_INIT, "emb", NULL, -1, EmbMethods,
    NULL, NULL, NULL, NULL
};

static PyObject*
PyInit_emb(void)
{
    return PyModule_Create(&EmbModule);
}
```

위의 코드를 main() 함수 바로 위에 삽입하십시오. 또한, Py\_Initialize()에 대한 호출 전에 다음 두 문장을 삽입하십시오:

```
numargs = argc;
PyImport_AppendInittab("emb", &PyInit_emb);
```

이 두 줄은 numargs 변수를 초기화하고, emb.numargs() 함수를 내장된 파이썬 인터프리터가 액세스할 수 있도록 만듭니다. 이러한 확장을 통해, 파이썬 스크립트는 다음과 같은 작업을 수행할 수 있습니다

```
import emb
print("Number of arguments", emb.numargs())
```

실제 응용 프로그램에서, 이 방법은 응용 프로그램의 API를 파이썬에 노출합니다.

### 3.1.5 C++로 파이썬 내장하기

파이썬을 C++ 프로그램에 내장하는 것도 가능합니다; 이것이 어떻게 수행되는지는 사용된 C++ 시스템의 세부 사항에 달려 있습니다; 일반적으로 C++로 메인 프로그램을 작성하고, C++ 컴파일러를 사용하여 프로그램을 컴파일하고 링크해야 합니다. C++을 사용하여 파이썬 자체를 다시 컴파일할 필요는 없습니다.

### 3.1.6 유닉스 계열 시스템에서 컴파일과 링크하기

파이썬 인터프리터를 응용 프로그램에 내장하기 위해 컴파일러(와 링커)에 적절한 플래그를 찾는 것이 늘 간단하지는 않습니다. 특히, 특히 파이썬이 자신에게 링크된 C 동적 확장(.so 파일)으로 구현된 라이브러리 모듈을 로드해야 하기 때문입니다.

필요한 컴파일러와 링커 플래그를 찾으려면, 설치 절차의 일부로 생성된 `pythonX.Y-config` 스크립트를 실행할 수 있습니다 (`python3-config` 스크립트도 사용 가능할 수 있습니다). 이 스크립트에는 여러 옵션이 있으며, 다음과 같은 것들은 여러분에 직접 유용할 것입니다:

- `pythonX.Y-config --cflags`는 컴파일 할 때의 권장 플래그를 제공합니다:

```
$ /opt/bin/python3.4-config --cflags
-I/opt/include/python3.4m -I/opt/include/python3.4m -DNDEBUG -g -fwrapv -O3 -
-Wall -Wstrict-prototypes
```

- `pythonX.Y-config --ldflags`는 링크 할 때의 권장 플래그를 제공합니다:

```
$ /opt/bin/python3.4-config --ldflags
-L/opt/lib/python3.4/config-3.4m -lpthread -ldl -lutil -lm -lpthon3.4m -
-Xlinker -export-dynamic
```

**참고:** 여러 파이썬 설치 간의 (특히 시스템 파이썬과 여러분이 직접 컴파일한 파이썬 간의) 혼란을 피하려면, 위의 예와 같이 `pythonX.Y-config`의 절대 경로를 사용하는 것이 좋습니다.

이 절차가 여러분을 위해 작동하지 않는다면 (모든 유닉스 계열 플랫폼에서 작동하는 것은 보장되지 않습니다; 하지만, 버그 보고를 환영합니다), 동적 링크에 관한 시스템의 설명서를 읽는 것과/이나 파이썬의 Makefile과(그 위치를 찾으려면 `sysconfig.get_makefile_filename()`를 사용하십시오) 컴파일 옵션을 검사해야 합니다. 이때, `sysconfig` 모듈은 여러분이 결합하려는 구성 값을 프로그래밍 방식으로 추출하는 데 유용한 도구입니다. 예를 들어:

```
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

## 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.
- Ellipsis 내장 상수.

**2to3** 파이썬 2.x 코드를 파이썬 3.x 코드로 변환하려고 시도하는 도구인데, 소스를 구문 분석하고 구문 분석 트리를 탐색해서 감지할 수 있는 대부분의 비호환성을 다룹니다.

2to3 는 표준 라이브러리에서 lib2to3 로 제공됩니다; 독립적으로 실행할 수 있는 스크립트는 Tools/scripts/2to3 로 제공됩니다. 2to3-reference 을 보세요.

**abstract base class (추상 베이스 클래스)** 추상 베이스 클래스는 `hasattr()` 같은 다른 테크닉들이 불편하거나 미묘하게 잘못된 (예를 들어, 매직 메서드) 경우, 인터페이스를 정의하는 방법을 제공함으로써 **덕 타이핑** 을 보완합니다. ABC 는 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 ABC 를 만들 수도 있습니다.

**annotation (어노테이션)** 관습에 따라 **형 힌트** 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 반환 값과 연결된 레이블입니다.

지역 변수의 어노테이션은 실행 시간에 액세스할 수 없지만, 전역 변수, 클래스 속성 및 함수의 어노테이션은 각각 모듈, 클래스, 함수의 `__annotations__` 특수 어트리뷰트에 저장됩니다.

이 기능을 설명하는 **변수 어노테이션**, **함수 어노테이션**, **PEP 484**, **PEP 526** 을 참조하세요.

**argument (인자)** 함수를 호출할 때 **함수** (또는 **메서드**) 로 전달되는 값. 두 종류의 인자가 있습니다:

- 키워드 인자 (**keyword argument**): 함수 호출 때 식별자가 앞에 붙은 인자 (예를 들어, `name=`) 또는 `**` 를 앞에 붙인 딕셔너리로 전달되는 인자. 예를 들어, 다음과 같은 `complex()` 호출에서 3 과 5 는 모두 키워드 인자입니다:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```



- 위치 인자 (*positional argument*): 키워드 인자가 아닌 인자. 위치 인자들은 인자 목록의 처음에 나오거나 **이터러블**의 앞에 \*를 붙여 전달할 수 있습니다. 예를 들어, 다음과 같은 호출에서 3과 5는 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 차이와 **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 이터레이터를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

비동기 제너레이터 함수는 `await` 표현식과, `async for` 문과, `async with` 문을 포함할 수 있습니다.

**asynchronous generator iterator (비동기 제너레이터 이터레이터)** 비동기 제너레이터 함수가 만드는 객체.

**비동기 이터레이터** 인데 `__anext__()`를 호출하면 어웨이터블 객체를 돌려주고, 이것은 다음 `yield` 표현식까지 비동기 제너레이터 함수의 바디를 실행합니다.

각 `yield`는 일시적으로 처리를 중단하고, 그 위치의 (지역 변수들과 대기 중인 `try`-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 이터레이터가 `__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 팝니다. **PEP 492**로 도입되었습니다.

**attribute (어트리뷰트)** 점표현식을 사용하는 이름으로 참조되는 객체와 결합한 값. 예를 들어, 객체 `o`가 어트리뷰트 `a`를 가지면, `o.a`처럼 참조됩니다.

**awaitable (어웨이터블)** `await` 표현식에 사용할 수 있는 객체. 코루틴이나 `__await__()` 메서드를 가진 객체가 될 수 있습니다. **PEP 492**를 보세요.

**BDFL** 자비로운 종신 독재자 (Benevolent Dictator For Life), 즉 Guido van Rossum, 파이썬의 창시자.

**binary file (바이너리 파일)** 바이트열류 객체들을 읽고 쓸 수 있는 파일 객체. 바이너리 파일의 예로는 바이너리 모드 ('rb', 'wb' 또는 'rb+')로 열린 파일, `sys.stdin.buffer`, `sys.stdout.buffer`, `io.BytesIO`와 `gzip.GzipFile`의 인스턴스를 들 수 있습니다.

`str` 객체를 읽고 쓸 수 있는 파일 객체에 대해서는 **텍스트 파일**도 참조하세요.

**bytes-like object (바이트열류 객체)** `bufferobjects`를 지원하고 C-연속 버퍼를 익스포트할 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.

**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.

**coercion (코어션)** 같은 형의 두 인자를 수반하는 연산이 일어나는 동안, 한 형의 인스턴스를 다른 형으로 묵시적으로 변환하는 것. 예를 들어, `int(3.15)`는 실수를 정수 3으로 변환합니다. 하지만, `3+4.5`에서, 각 인자는 다른 형이고 (하나는 `int`, 다른 하나는 `float`), 둘을 더하기 전에 같은 형으로 변환해야 합니다. 그렇지 않으면 `TypeError`를 일으킵니다. 코어션 없이는, 호환되는 형들조차도 프로그래머가 같은 형으로 정규화해주어야 합니다, 예를 들어, 그냥 `3+4.5` 하는 대신 `float(3)+4.5`.

**complex number (복소수)** 익숙한 실수 시스템의 확장인데, 모든 숫자가 실수부와 허수부의 합으로 표현됩니다. 허수부는 실수에 허수 단위 (-1의 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 필요하다면, `cmath`를 사용합니다. 복소수의 활용은 꽤 수준 높은 수학적 기능입니다. 필요하다고 느끼지 못한다면, 거의 확실히 무시해도 좋습니다.

**context manager (컨텍스트 관리자)** `__enter__()`와 `__exit__()` 메서드를 정의함으로써 `with` 문에서 보이는 환경을 제어하는 객체. **PEP 343**으로 도입되었습니다.

**context variable (컨텍스트 변수)** 컨텍스트에 따라 다른 값을 가질 수 있는 변수. 이는 각 실행 스레드가 변수에 대해 다른 값을 가질 수 있는 스레드-로컬 저장소와 비슷합니다. 그러나, 컨텍스트 변수를 통해, 하나의 실행 스레드에 여러 컨텍스트가 있을 수 있으며 컨텍스트 변수의 주 용도는 동시성 비동기 태스크에서 변수를 추적하는 것입니다. `contextvars`를 참조하십시오.

**contiguous (연속)** 버퍼는 정확히 C-연속 (*C-contiguous*)이거나 포트란 연속 (*Fortran contiguous*)일 때 연속이라고 여겨집니다. 영차원 버퍼는 C-연속이면서 포트란 연속입니다. 일차원 배열에서, 항목들은 서로에 인접하고, 0에서 시작하는 오름차순 인덱스의 순서대로 메모리에 배치되어야 합니다. 다차원 C-연속 배열에서, 메모리 주소의 순서대로 항목들을 방문할 때 마지막 인덱스가 가장 빨리 변합니다. 하지만, 포트란 연속 배열에서는, 첫 번째 인덱스가 가장 빨리 변합니다.

**coroutine (코루틴)** 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.org](https://python.org)에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 《CPython》이 사용됩니다.

**decorator (데코레이터)** 다른 함수를 돌려주는 함수인데, 보통 `@wrapper` 문법을 사용한 함수 변환으로 적용됩니다. 데코레이터의 흔한 예는 `classmethod()`과 `staticmethod()`입니다.

데코레이터 문법은 단지 편의 문법일 뿐입니다. 다음 두 함수 정의는 의미상으로 동등합니다:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def f(...):
    ...
```

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의와 클래스 정의의 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 디렉터리에서 `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 `descriptors`에 나옵니다.

**dictionary (딕셔너리)** 임의의 키를 값에 대응시키는 연관 배열 (associative array). 키는 `__hash__()` 와 `__eq__()` 메서드를 갖는 모든 객체가 될 수 있습니다. 필에서 해시라고 부릅니다.

**dictionary comprehension** A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See comprehensions.

**dictionary view (딕셔너리 뷰)** `dict.keys()`, `dict.values()`, `dict.items()` 메서드가 돌려주는 객체들을 딕셔너리 뷰라고 부릅니다. 이것들은 딕셔너리 항목들에 대한 동적인 뷰를 제공하는데, 딕셔너리가 변경될 때, 뷰가 이 변화를 반영한다는 뜻입니다. 딕셔너리 뷰를 완전한 리스트로 바꾸려면 `list(dictview)`를 사용하면 됩니다. `dict-views`를 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 꺾꺾댄다면, 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 클래스로 보완될 수 있음에 유의해야 합니다.) 대신에, `hasattr()` 검사나 [EAFP](#) 프로그래밍을 씁니다.

**EAFP** 허락보다는 용서를 구하기가 쉽다 (Easier to ask for forgiveness than permission). 이 흔히 볼 수 있는 파이썬 코딩 스타일은, 올바른 키나 어트리뷰트의 존재를 가정하고, 그 가정이 틀리면 예외를 잡습니다. 이 깔끔하고 빠른 스타일은 많은 `try`와 `except` 문의 존재로 특징지어집니다. 이 테크닉은 C와 같은 다른 많은 언어에서 자주 사용되는 [LBYL](#) 스타일과 대비됩니다.

**expression (표현식)** 어떤 값으로 구해질 수 있는 문법적인 조각. 다른 말로 표현하면, 표현식은 리터럴, 이름, 어트리뷰트 액세스, 연산자, 함수들과 같은 값을 돌려주는 표현 요소들을 쌓아 올린 것입니다. 다른 많은 언어와 대조적으로, 모든 언어 구성물들이 표현식인 것은 아닙니다. `while`처럼, 표현식으로 사용할 수 없는 **문장**들이 있습니다. 대입 또한 문장이고, 표현식이 아닙니다.

**extension module (확장 모듈)** C 나 C++로 작성된 모듈인데, 파이썬의 C API를 사용해서 핵심이나 사용자 코드와 상호 작용합니다.

**f-string (f-문자열)** 'f' 나 'F' 를 앞에 붙인 문자열 리터럴들을 흔히 《f-문자열》이라고 부르는데, 포맷 문자열 리터럴의 줄임말입니다. [PEP 498](#) 을 보세요.

**file object (파일 객체)** 하부 자원에 대해 파일 지향적 API(`read()` 나 `write()` 같은 메서드들)를 드러내는 객체. 만들어진 방법에 따라, 파일 객체는 실제 디스크 상의 파일이나 다른 저장 장치나 통신 장치 (예를 들어, 표준 입출력, 인-메모리 버퍼, 소켓, 파이프, 등등)에 대한 액세스를 중계할 수 있습니다. 파일 객체는 파일류 객체 (*file-like objects*)나 스트림 (*streams*) 이라고도 불립니다.

실제로는 세 부류의 파일 객체들이 있습니다. 날(*raw*) **바이너리 파일**, 버퍼드(*buffered*) **바이너리 파일**, **텍스트 파일**. 이들의 인터페이스는 `io` 모듈에서 정의됩니다. 파일 객체를 만드는 규범적인 방법은 `open()` 함수를 쓰는 것입니다.

**file-like object (파일류 객체)** **파일 객체** 의 비슷한 말.

**finder (파인더)** 임포트될 모듈을 위한 로더를 찾으려고 시도하는 객체.

파이썬 3.3. 이후로, 두 종류의 파인더가 있습니다: `sys.meta_path`와 함께 사용하는 메타 경로 파인더와 `sys.path_hooks`와 함께 사용하는 경로 엔트리 파인더.

더 자세한 내용은 [PEP 302](#), [PEP 420](#), [PEP 451](#)에 나옵니다.

**floor division (정수 나눗셈)** 가장 가까운 정수로 내림하는 수학적 나눗셈. 정수 나눗셈 연산자는 `//` 다. 예를 들어, 표현식 `11 // 4`의 값은 2가 되지만, 실수 나눗셈은 2.75를 돌려줍니다. `(-11) // 4`가 -2.75를 내림한 -3이 됨에 유의해야 합니다. [PEP 238](#)을 보세요.

**function (함수)** 호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 인자가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. 매개변수와 메서드와 function 섹션도 보세요.

**function annotation (함수 어노테이션)** 함수 매개변수나 반환 값의 어노테이션.

함수 어노테이션은 일반적으로 형 힌트로 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 어노테이션과 [PEP 484](#)를 참조하세요.

**\_\_future\_\_** 프로그래머가 현재 인터프리터와 호환되지 않는 새 언어 기능들을 활성화할 수 있도록 하는 가상 모듈.

`__future__` 모듈을 임포트하고 그 변수들의 값들을 구해서, 새 기능이 언제 처음으로 언어에 추가되었고, 언제부터 그것이 기본이 되는지 볼 수 있습니다:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

**garbage collection (가비지 수거)** 더 사용되지 않는 메모리를 반납하는 절차. 파이썬은 참조 횟수 추적과 참조 순환을 감지하고 끊을 수 있는 순환 가비지 수거기를 통해 가비지 수거를 수행합니다. 가비지 수거기는 `gc` 모듈을 사용해서 제어할 수 있습니다.

**generator (제너레이터)** 제너레이터 이터레이터를 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 이터레이터를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

**generator iterator (제너레이터 이터레이터)** 제너레이터 함수가 만드는 객체.

각 `yield`는 일시적으로 처리를 중단하고, 그 위치의 (지역 변수들과 대기 중인 `try`-문들을 포함하는) 실행 상태를 기억합니다. 제너레이터 이터레이터가 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 표현식처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

**generic function (제네릭 함수)** 같은 연산을 서로 다른 형들에 대해 구현한 여러 함수로 구성된 함수. 호출 때 어떤 구현이 사용될지는 디스패치 알고리즘에 의해 결정됩니다.

싱글 디스패치 용어집 항목과 `functools.singledispatch()` 데코레이터와 [PEP 443](#)도 보세요.

**GIL** 전역 인터프리터 록을 보세요.

**global interpreter lock (전역 인터프리터 록)** 한 번에 오직 하나의 스레드가 파이썬 **바이트 코드**를 실행하도록 보장하기 위해 **CPython** 인터프리터가 사용하는 메커니즘. (dict와 같은 중요한 내장형들을 포함하는) 객체 모델이 묵시적으로 동시 액세스에 대해 안전하도록 만들어서 CPython 구현을 단순하게 만듭니다. 인터프리터 전체를 잠그는 것은 인터프리터를 다중스레드화하기 쉽게 만드는 대신, 다중 프로세서 기계가 제공하는 병렬성의 많은 부분을 희생합니다.

하지만, 어떤 확장 모듈들은, 표준이나 제삼자 모두, 압축이나 해싱 같은 계산 집약적인 작업을 수행할 때는 GIL을 반납하도록 설계되었습니다. 또한, I/O를 할 때는 항상 GIL을 반납합니다.

(훨씬 더 미세하게 공유 데이터를 잠그는) 《스레드에 자유로운 (free-threaded)》 인터프리터를 만들고자 하는 과거의 노력은 성공적이지 못했는데, 혼란 단일 프로세서 경우의 성능 저하가 심하기 때문입니다. 이 성능 이슈를 극복하는 것은 구현을 훨씬 복잡하게 만들어서 유지 비용이 더 들어갈 것으로 여겨지고 있습니다.

**hash-based pyc (해시 기반 pyc)** 유효성을 판별하기 위해 해당 소스 파일의 최종 수정 시간이 아닌 해시를 사용하는 바이트 코드 캐시 파일. **pyc-invalidation**을 참조하세요.

**hashable (해시 가능)** 객체가 일생 그 값이 변하지 않는 해시값을 갖고 (`__hash__()` 메서드가 필요합니다), 다른 객체와 비교될 수 있으면 (`__eq__()` 메서드가 필요합니다), 해시 가능하다고 합니다. 같다고 비교되는 해시 가능한 객체들의 해시값은 같아야 합니다.

해시 가능성은 객체를 딕셔너리의 키나 집합의 멤버로 사용할 수 있게 하는데, 이 자료 구조들이 내부적으로 해시값을 사용하기 때문입니다.

대부분 파이썬의 불변 내장 객체들은 해시 가능합니다; (리스트나 딕셔너리 같은) 가변 컨테이너들은 그렇지 않습니다; (튜플이나 frozenset 같은) 불변 컨테이너들은 그들의 요소들이 해시 가능할 때만 해시 가능합니다. 사용자 정의 클래스의 인스턴스 객체들은 기본적으로 해시 가능합니다. (자기 자신을 제외하고는) 모두 다르다고 비교되고, 해시값은 `id()`로 부터 만들어집니다.

**IDLE** 파이썬을 위한 통합 개발 환경 (Integrated Development Environment). IDLE은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

**immutable (불변)** 고정된 값을 갖는 객체. 불변 객체는 숫자, 문자열, 튜플을 포함합니다. 이런 객체들은 변경될 수 없습니다. 새 값을 저장하려면 새 객체를 만들어야 합니다. 변하지 않는 해시값이 있어야 하는 곳에서 중요한 역할을 합니다, 예를 들어, 딕셔너리의 키.

**import path (임포트 경로)** **경로 기반 파인더**가 임포트 할 모듈을 찾기 위해 검색하는 장소들 (또는 **경로 엔트리**)의 목록. 임포트 하는 동안, 이 장소들의 목록은 보통 `sys.path`로부터 옵니다, 하지만 서브패키지의 경우 부모 패키지의 `__path__` 어트리뷰트로부터 올 수도 있습니다.

**importing (임포트)** 한 모듈의 파이썬 코드가 다른 모듈의 파이썬 코드에서 사용될 수 있도록 하는 절차.

**importer (임porter)** 모듈을 찾기도 하고 로드 하기도 하는 객체; 동시에 **파인더**이자 **로더** 객체입니다.

**interactive (대화형)** 파이썬은 대화형 인터프리터를 갖고 있는데, 인터프리터 프롬프트에서 문장과 표현식을 입력할 수 있고, 즉각 실행된 결과를 볼 수 있다는 뜻입니다. 인자 없이 단지 `python`을 실행하세요 (컴퓨터의 주메뉴에서 선택하는 것도 가능할 수 있습니다). 새 아이디어를 검사하거나 모듈과 패키지를 들여다보는 매우 강력한 방법입니다 (`help(x)`를 기억하세요).

**interpreted (인터프리티드)** 바이트 코드 컴파일러의 존재 때문에 그 부분이 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 잡아줄 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.

**key function (키 함수)** 키 함수 또는 콜레이션(`collation`) 함수는 정렬(`sorting`)이나 배열(`ordering`)에 사용되는 값을 돌려주는 콜러블입니다. 예를 들어, `locale.strxfrm()`은 로케일 특정 방식을 따르는 정렬 키를 만드는 데 사용됩니다.

파이썬의 많은 도구가 요소들이 어떻게 순서 지어지고 묶이는지를 제어하기 위해 키 함수를 받아들입니다. 이런 것들에는 `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()`이 있습니다.

키 함수를 만드는 데는 여러 방법이 있습니다. 예를 들어, `str.lower()` 메서드는 케이스 구분 없는 정렬을 위한 키 함수로 사용될 수 있습니다. 대안적으로, 키 함수는 `lambda` 표현식으로 만들 수도 있는데, 이런 식입니다: `lambda r: (r[0], r[2])`. 또한, `operator` 모듈은 세 개의 키 함수 생성자를 제공합니다: `attrgetter()`, `itemgetter()`, `methodcaller()`. 키 함수를 만들고 사용하는 법에 대한 예로 **Sorting HOW TO**를 보세요.

**keyword argument (키워드 인자)** **인자**를 보세요.

**lambda (람다)** 호출될 때 값이 구해지는 하나의 **표현식**으로 구성된 이름 없는 인라인 함수. 람다 함수를 만드는 문법은 `lambda [parameters]: expression`입니다.

**LBYL** 뚝기 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 《뚝기》 간에 경쟁 조건을 만들게 될 위험이 있습니다. 예를 들어, 코드 `if key in mapping: return mapping[key]`는 검사 후에, 하지만 조회 전에, 다른 스레드가 `key`를 `mapping`에서 제거하면 실패할 수 있습니다. 이런 이슈는 록이나 EAFP 접근법을 사용함으로써 해결될 수 있습니다.

**list (리스트)** 내장 파이썬 **시퀀스**. 그 이름에도 불구하고, 원소에 대한 액세스가  $O(1)$ 이기 때문에, 연결 리스트(linked list)보다는 다른 언어의 배열과 유사합니다.

**list comprehension (리스트 컴프리헨션)** 시퀀스의 요소들 전부 또는 일부를 처리하고 그 결과를 리스트로 돌려주는 간결한 방법. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]`는 0에서 255 사이에 있는 짝수들의 16진수(0x..)들을 포함하는 문자열의 리스트를 만듭니다. if 절은 생략할 수 있습니다. 생략하면, `range(256)`에 있는 모든 요소가 처리됩니다.

**loader (로더)** 모듈을 로드하는 객체. `load_module()`이라는 이름의 메서드를 정의해야 합니다. 로더는 보통 **과인더**가 돌려줍니다. 자세한 내용은 **PEP 302**를, 추상 베이스 클래스는 `importlib.abc.Loader`를 보세요.

**magic method (매직 메서드)** 특수 메서드의 비공식적인 비슷한 말.

**mapping (매핑)** 임의의 키 조회를 지원하고 Mapping이나 MutableMapping 추상 베이스 클래스에 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 디렉터리, 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.

metaclasses에서 더 자세한 내용을 찾을 수 있습니다.

**method (메서드)** 클래스 바디 안에서 정의되는 함수. 그 클래스의 인스턴스의 어트리뷰트로서 호출되면, 그 메서드는 첫 번째 **인자**(보통 `self`라고 불린다)로 인스턴스 객체를 받습니다. **함수**와 **중첩된 스코프**를 보세요.

**method resolution order (메서드 결정 순서)** 메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서입니다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 [The Python 2.3 Method Resolution Order](#)를 보면 됩니다.

**module (모듈)** 파이썬 코드의 조직화 단위를 담당하는 객체. 모듈은 임의의 파이썬 객체들을 담는 이름 공간을 갖습니다. 모듈은 **임포트** 절차에 의해 파이썬으로 로드됩니다.

패키지도 보세요.

**module spec (모듈 스펙)** 모듈을 로드하는데 사용되는 임포트 관련 정보들을 담고 있는 이름 공간. `importlib.machinery.ModuleSpec`의 인스턴스.

**MRO** 메서드 결정 순서를 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 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 디렉터리로 구현됩니다. 객체에 중첩된 이름 공간(메서드에서) 뿐만 아니라 지역, 전역, 내장 이름 공간이 있습니다. 이름 공간은 이름 충돌을 방지해서 모듈성을 지원합니다. 예를 들어, 함수 `builtins.open`과 `os.open()`은 그들의 이름 공간에 의해 구별됩니다. 또한, 이름 공간은 어떤 모듈이 함수를 구현하는지를 분명하게 만들어서 가독성과 유지 보수성에 도움을 줍니다. 예를 들어, `random.seed()` 또는 `itertools.islice()`라고 쓰면 그 함수들이 각각 `random`과 `itertools` 모듈에 의해 구현되었음이 명확해집니다.

**namespace package (이름 공간 패키지)** 오직 서브 패키지들의 컨테이너로만 기능하는 **PEP 420 패키지**. 이름 공간 패키지는 물리적인 실체가 없을 수도 있고, 특히 `__init__.py` 파일이 없으므로 정규 패키지와는 다릅니다.

모듈도 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, `__getattr__()`, 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.

**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 혹) 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 혹)** `sys.path_hook` 리스트에 있는 콜러블인데, 특정 경로 엔트리 에서 모듈을 찾는 법을 알고 있다면 경로 엔트리 파인더 를 돌려줍니다.

**path based finder** (경로 기반 파인더) 기본 메타 경로 파인더들 중 하나인데, **임포트 경로** 에서 모듈을 찾습니다.

**path-like object** (경로류 객체) 파일 시스템 경로를 나타내는 객체. 경로류 객체는 경로를 나타내는 `str` 나 `bytes` 객체가거나 `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__
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

모듈을 가리키는데 사용될 때, 완전히 정규화된 이름 (*fully qualified name*)은 모든 부모 패키지들을 포함해서 모듈로 가는 점으로 분리된 이름을 의미합니다, 예를 들어, `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

**reference count** (참조 횟수) 객체에 대한 참조의 개수. 객체의 참조 횟수가 0으로 떨어지면, 메모리가 반납됩니다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소입니다. `sys` 모듈은 특정 객체의 참조 횟수를 돌려주는 `getrefcount()` 을 정의합니다.

**regular package** (정규 패키지) `__init__.py` 파일을 포함하는 디렉터리와 같은 전통적인 패키지.

이름 공간 패키지 도 보세요.

**\_\_slots\_\_** 클래스 내부의 선언인데, 인스턴스 어트리뷰트들을 위한 공간을 미리 선언하고 인스턴스 디스너리를 제거함으로써 메모리를 절감하는 효과를 줍니다. 인기 있기는 하지만, 이 테크닉은 올바르게 사용하기가 좀 까다로운 편이라서, 메모리에 민감한 응용 프로그램에서 많은 수의 인스턴스가 있는 특별한 경우로 한정하는 것이 좋습니다.

**sequence** (시퀀스) `__getitem__()` 특수 메서드를 통해 정수 인덱스를 사용한 빠른 요소 액세스를 지원하고, 시퀀스의 길이를 돌려주는 `__len__()` 메서드를 정의하는 **이터러블**. 몇몇 내장 시퀀스들을 나열해보면, `list`, `str`, `tuple`, `bytes` 가 있습니다. `dict` 또한 `__getitem__()` 과 `__len__()` 을 지원하지만, 조회에 정수 대신 임의의 불변 키를 사용하기 때문에 시퀀스가 아니라 매핑으로 취급된다는 것에 주의해야 합니다.

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()`.

**set comprehension** A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

**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.

**special method** (특수 메서드) 파이썬이 형에 어떤 연산을, 덧셈 같은, 실행할 때 묵시적으로 호출되는 메서드. 이런 메서드는 두 개의 밑줄로 시작하고 끝나는 이름을 갖고 있습니다. 특수 메서드는 `specialnames` 에 문서로 만들어져 있습니다.

**statement** (문장) 문장은 스위트 (코드의 《블록 (block)》) 를 구성하는 부분입니다. 문장은 **표현식** 이거나 키워드를 사용하는 여러 가지 구조물 중의 하나입니다. 가령 `if`, `while`, `for`.

**text encoding** (텍스트 인코딩) 유니코드 문자열을 바이트열로 인코딩하는 코덱.

**text file** (텍스트 파일) `str` 객체를 읽고 쓸 수 있는 **파일 객체**. 종종, 텍스트 파일은 실제로는 바이트 지향 데이터스트림을 액세스하고 **텍스트 인코딩** 을 자동 처리합니다. 텍스트 파일의 예로는 텍스트 모드 ('r' 또는 'w') 로 열린 파일, `sys.stdin`, `sys.stdout`, `io.StringIO` 의 인스턴스를 들 수 있습니다.

**바이트열류 객체** 를 읽고 쓸 수 있는 파일 객체에 대해서는 **바이너리 파일** 도 참조하세요.

**triple-quoted string** (삼중 따옴표 된 문자열) 따옴표 (《) 나 작은따옴표 (〈) 세 개로 둘러싸인 문자열. 그냥 따옴표 하나로 둘러싸인 문자열에 없는 기능을 제공하지는 않지만, 여러 가지 이유에서 쓸모가 있습니다. 이스케이프 되지 않은 작은따옴표나 큰따옴표를 문자열 안에 포함할 수 있도록 하고, 연결 문자를 쓰지 않고도 여러 줄에 걸쳐 쓸 수 있는데, 독스트링을 쓸 때 특히 쓸모 있습니다.

**type** (형) 파이썬 객체의 형은 그것이 어떤 종류의 객체인지를 결정합니다; 모든 객체는 형이 있습니다. 객체의 형은 `__class__` 어트리뷰트로 액세스할 수 있거나 `type(obj)` 로 얻을 수 있습니다.

**type alias** (형 에일리어스) 형을 식별자에 대입하여 만들어지는 형의 동의어.

형 에일리어스는 형 힌트를 단순화하는 데 유용합니다. 예를 들면:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

이 기능을 설명하는 `typing`과 **PEP 484**를 참조하세요.

**type hint** (형 힌트) 변수, 클래스 어트리뷰트 및 함수 매개변수 나 반환 값의 기대되는 형을 지정하는 어노테이션.

형 힌트는 선택 사항이며 파이썬에서 강제되지는 않습니다. 하지만, 정적 형 분석 도구에 유용하며 IDE의 코드 완성 및 리팩토링을 돕습니다.

지역 변수를 제외하고, 전역 변수, 클래스 어트리뷰트 및 함수의 형 힌트는 `typing.get_type_hints()` 를 사용하여 액세스할 수 있습니다.

이 기능을 설명하는 `typing`과 **PEP 484**를 참조하세요.

**universal newlines** (유니버설 줄 넘김) 다음과 같은 것들을 모두 줄의 끝으로 인식하는, 텍스트 스트림을 해석하는 태도: 유닉스 개행 문자 관례 `'\n'`, 윈도우즈 관례 `'\r\n'`, 예전의 매킨토시 관례 `'\r'`. 추가적인 사용에 관해서는 `bytes.splitlines()` 뿐만 아니라 **PEP 278** 와 **PEP 3116** 도 보세요.

**variable annotation** (변수 어노테이션) 변수 또는 클래스 어트리뷰트의 어노테이션.

변수 또는 클래스 어트리뷰트에 어노테이션을 달 때 대입은 선택 사항입니다:

```
class C:
    field: 'annotation'
```

변수 어노테이션은 일반적으로 형 힌트로 사용됩니다: 예를 들어, 이 변수는 `int` 값을 가질 것으로 기대됩니다:

```
count: int = 0
```

변수 어노테이션 문법은 섹션 `annassign` 에서 설명합니다.

이 기능을 설명하는 함수 어노테이션, **PEP 484** 및 **PEP 526**을 참조하세요.

**virtual environment** (가상 환경) 파이썬 사용자와 응용 프로그램이, 같은 시스템에서 실행되는 다른 파이썬 응용 프로그램들의 동작에 영향을 주지 않으면서, 파이썬 배포 패키지들을 설치하거나 업그레이드 하는 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 코드](#)를 실행합니다.

**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

---

이 설명서는 `reStructuredText` 소스에서 만들어진 것으로, 파이썬 설명서를 위해 특별히 제작된 문서 처리기인 `Sphinx` 를 사용했습니다.

설명서와 이를 위한 툴체인 개발은 파이썬 자체와 마찬가지로 전적으로 자원봉사자의 노력입니다. 기여하고 싶다면, 참여 방법에 대한 정보는 `reporting-bugs` 페이지를 참고하십시오. 새로운 자원봉사자는 언제나 환영합니다!

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 작가;
- `reStructuredText`와 `Docutils` 스위트를 만드는 `Docutils` 프로젝트.
- Fredrik Lundh, 그의 `Alternative Python Reference` 프로젝트에서 `Sphinx`가 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!



## 역사와 라이선스

## C.1 소프트웨어의 역사

파이썬은 ABC라는 언어의 후계자로서 네덜란드의 Stichting Mathematisch Centrum (CWI, <https://www.cwi.nl/> 참조)의 Guido van Rossum에 의해 1990년대 초반에 만들어졌습니다. 파이썬에는 다른 사람들의 많은 공헌이 포함되었지만, Guido는 파이썬의 주요 저자로 남아 있습니다.

1995년, Guido는 Virginia의 Reston에 있는 Corporation for National Research Initiatives(CNRI, <https://www.cnri.reston.va.us/> 참조)에서 파이썬 작업을 계속했고, 이곳에서 여러 버전의 소프트웨어를 출시했습니다.

2000년 5월, Guido와 파이썬 핵심 개발팀은 BeOpen.com으로 옮겨서 BeOpen PythonLabs 팀을 구성했습니다. 같은 해 10월, PythonLabs 팀은 Digital Creations(현재 Zope Corporation; <https://www.zope.org/> 참조)로 옮겼습니다. 2001년, 파이썬 소프트웨어 재단(PSF, <https://www.python.org/psf/> 참조)이 설립되었습니다. 이 단체는 파이썬 관련 지적 재산을 소유하도록 특별히 설립된 비영리 조직입니다. Zope Corporation은 PSF의 후원 회원입니다.

모든 파이썬 배포판은 공개 소스입니다(공개 소스 정의에 대해서는 <https://opensource.org/>를 참조하십시오). 역사적으로, 대부분(하지만 전부는 아닙니다) 파이썬 배포판은 GPL과 호환됩니다; 아래의 표는 다양한 배포판을 요약한 것입니다.

배포판	파생된 곳	해	소유자	GPL 호환?
0.9.0 ~ 1.2	n/a	1991-1995	CWI	yes
1.3 ~ 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 이상	2.1.1	2001-현재	PSF	yes

**참고:** GPL과 호환된다는 것은 우리가 GPL로 파이썬을 배포한다는 것을 의미하지는 않습니다. 모든 파이썬 라이선스는 GPL과 달리 여러분의 변경을 공개 소스로 만들지 않고 수정된 버전을 배포할 수 있게

합니다. GPL 호환 라이선스는 파이썬과 GPL 하에 발표된 다른 소프트웨어를 결합할 수 있게 합니다; 다른 것들은 그렇지 않습니다.

---

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

## C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

파이썬 소프트웨어와 설명서는 *PSF License Agreement*에 따라 라이선스가 부여됩니다.

파이썬 3.8.6부터, 설명서의 예제, 조리법 및 기타 코드는 PSF License Agreement와 *Zero-Clause BSD license*에 따라 이중 라이선스가 부여됩니다.

파이썬에 통합된 일부 소프트웨어에는 다른 라이선스가 적용됩니다. 라이선스는 해당 라이선스에 해당하는 코드와 함께 나열됩니다. 이러한 라이선스의 불완전한 목록은 포함된 소프트웨어에 대한 라이선스 및 승인을 참조하십시오.

### C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.8.13

1. This LICENSE AGREEMENT is between the Python Software Foundation,  
→("PSF"), and  
the Individual or Organization ("Licensee") accessing and otherwise  
→using Python  
3.8.13 software in source or binary form and its associated  
→documentation.
2. Subject to the terms and conditions of this License Agreement, PSF  
→hereby  
grants Licensee a nonexclusive, royalty-free, world-wide license to  
→reproduce,  
analyze, test, perform and/or display publicly, prepare derivative  
→works,  
distribute, and otherwise use Python 3.8.13 alone or in any derivative  
version, provided, however, that PSF's License Agreement and PSF's  
→notice of  
copyright, i.e., "Copyright © 2001-2022 Python Software Foundation; All  
→Rights  
Reserved" are retained in Python 3.8.13 alone or in any derivative  
→version  
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or  
incorporates Python 3.8.13 or any part thereof, and wants to make the  
derivative work available to others as provided herein, then Licensee  
→hereby  
agrees to include in any such work a brief summary of the changes made  
→to Python  
3.8.13.
4. PSF is making Python 3.8.13 available to Licensee on an "AS IS" basis.  
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY  
→OF  
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY  
→REPRESENTATION OR  
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR  
→THAT THE  
USE OF PYTHON 3.8.13 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.



5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.13 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.13, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.8.13, 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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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 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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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 3.8.13 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 라이선스 및 승인

이 섹션은 파이썬 배포판에 포함된 제삼자 소프트웨어에 대한 불완전하지만 늘어나고 있는 라이선스와 승인의 목록입니다.

### C.3.1 메르센 트위스터

`_random` 모듈은 <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html> 에서 내려받은 코드에 기반한 코드를 포함합니다. 다음은 원래 코드의 주석을 그대로 옮긴 것입니다:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote
   products derived from this software without specific prior written
   permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

### C.3.2 소켓

socket 모듈은 `getaddrinfo()`와 `getnameinfo()` 함수를 사용합니다. 이들은 WIDE Project, <http://www.wide.ad.jp/>, 에서 온 별도 소스 파일로 코딩되어 있습니다.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.3 비동기 소켓 서비스

asynchat과 asyncore 모듈은 다음과 같은 주의 사항을 포함합니다:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.4 쿠키 관리

http.cookies 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

    All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

### C.3.5 실행 추적

trace 모듈은 다음과 같은 주의 사항을 포함합니다:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

### C.3.6 UUencode 및 UUdecode 함수

uu 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

### C.3.7 XML 원격 프로시저 호출

xmlrpc.client 모듈은 다음과 같은 주의 사항을 포함합니다:

```
The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

### C.3.8 test\_epoll

test\_epoll 모듈은 다음과 같은 주의 사항을 포함합니다:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### C.3.9 Select kqueue

select 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



### C.3.10 SipHash24

파일 `Python/pyhash.c` 에는 Dan Bernstein의 SipHash24 알고리즘의 Marek Majkowski의 구현이 포함되어 있습니다. 여기에는 다음과 같은 내용이 포함되어 있습니다:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
  https://github.com/majek/csiphash/

Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/siphash24/little)
  djb (supercop/crypto_auth/siphash24/little2)
  Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

### C.3.11 strtod 와 dtoa

C double과 문자열 간의 변환을 위한 C 함수 `dtoa` 와 `strtod` 를 제공하는 파일 `Python/dtoa.c` 는 현재 <http://www.netlib.org/fp/> 에서 얻을 수 있는 David M. Gay의 같은 이름의 파일에서 파생되었습니다. 2009년 3월 16일에 받은 원본 파일에는 다음과 같은 저작권 및 라이선스 공지가 포함되어 있습니다:

```
/*****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *****/
```

### C.3.12 OpenSSL

모듈 `hashlib`, `posix`, `ssl`, `crypt` 는 운영 체제가 사용할 수 있게 하면 추가의 성능을 위해 **OpenSSL** 라이브러리를 사용합니다. 또한, 윈도우와 맥 OS X 파이썬 설치 프로그램은 **OpenSSL** 라이브러리 사본을 포함할 수 있으므로, 여기에 **OpenSSL** 라이선스 사본을 포함합니다:

```
LICENSE ISSUES
=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
-----

/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 * endorse or promote products derived from this software without
 * prior written permission. For written permission, please contact
 * openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 * nor may "OpenSSL" appear in their names without prior written
 * permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 * acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
 * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * 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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com). This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/
```

Original SSLeay License

-----

```
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
* notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in the
* documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
* must display the following acknowledgement:
* "This product includes cryptographic software written by
* Eric Young (eay@cryptsoft.com)"
* The word 'cryptographic' can be left out if the rouines from the library
* being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
* the apps directory (application code) you must include an acknowledgement:
* "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

### C.3.13 expat

pyexpat 확장은 빌드를 --with-system-expat 로 구성하지 않는 한, 포함된 expat 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

### C.3.14 libffi

\_ctypes 확장은 빌드를 --with-system-libffi 로 구성하지 않는 한, 포함된 libffi 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

### C.3.15 zlib

zlib 확장은 시스템에서 발견된 zlib 버전이 너무 오래되어서 빌드에 사용될 수 없으면, 포함된 zlib 소스 사본을 사용하여 빌드됩니다:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler
```

```
This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

```
Jean-loup Gailly      Mark Adler
jloup@gzip.org
```

```
Mark Adler
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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

### C.3.17 libmpdec

\_decimal 모듈은 빌드를 --with-system-libmpdec 로 구성하지 않는 한, 포함된 libmpdec 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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

test 패키지의 C14N 2.0 테스트 스위트(Lib/test/xmltestdata/c14n-20/)는 W3C 웹 사이트 <https://www.w3.org/TR/xml-c14n2-testcases/> 에서 가져왔으며 3-절 BSD 라이선스 하에 배포됩니다:

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, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.





## APPENDIX D

---

### 저작권

---

파이썬과 이 설명서는:

Copyright © 2001-2022 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 라이선스 및 사용 권한 정보는 [역사와 라이선스](#) 에서 제공합니다.



## Non-alphabetical

..., 67  
 2to3, 67  
 >>>, 67  
 \_\_future\_\_, 71  
 \_\_slots\_\_, 77

## A

abstract base class (추상 베이스 클래스), 67  
 annotation (어노테이션), 67  
 argument (인자), 67  
 asynchronous context manager (비동기 컨텍스트 관리자), 68  
 asynchronous generator (비동기 제너레이터), 68  
 asynchronous generator iterator (비동기 제너레이터 이터레이터), 68  
 asynchronous iterable (비동기 이터러블), 68  
 asynchronous iterator (비동기 이터레이터), 68  
 attribute (어트리뷰트), 68  
 awaitable (어웨이터블), 68

## B

BDFL, 68  
 binary file (바이너리 파일), 68  
 bytecode (바이트 코드), 69  
 bytes-like object (바이트열류 객체), 68

## C

callback (콜백), 69  
 C-contiguous, 69  
 class (클래스), 69  
 class variable (클래스 변수), 69  
 coercion (코어션), 69  
 complex number (복소수), 69  
 context manager (컨텍스트 관리자), 69  
 context variable (컨텍스트 변수), 69  
 contiguous (연속), 69  
 coroutine (코루틴), 69  
 coroutine function (코루틴 함수), 69  
 CPython, 69

## D

deallocation, object, 48  
 decorator (데코레이터), 69  
 descriptor (디스크립터), 70  
 dictionary (딕셔너리), 70  
 dictionary comprehension, 70  
 dictionary view (딕셔너리 뷰), 70  
 docstring (독스트링), 70  
 duck-typing (덕 타이핑), 70

## E

EAFP, 70  
 expression (표현식), 70  
 extension module (확장 모듈), 70

## F

f-string (f-문자열), 70  
 file object (파일 객체), 70  
 file-like object (파일류 객체), 70  
 finalization, of objects, 48  
 finder (파인더), 71  
 floor division (정수 나눗셈), 71  
 Fortran contiguous, 69  
 function (함수), 71  
 function annotation (함수 어노테이션), 71

## G

garbage collection (가비지 수거), 71  
 generator, 71  
 generator (제너레이터), 71  
 generator expression, 71  
 generator expression (제너레이터 표현식), 71  
 generator iterator (제너레이터 이터레이터), 71  
 generic function (제네릭 함수), 71  
 GIL, 71  
 global interpreter lock (전역 인터프리터 록), 72

## H

hash-based pyc (해시 기반 pyc), 72  
 hashable (해시 가능), 72

## I

IDLE, [72](#)  
 immutable (불변), [72](#)  
 import path (임포트 경로), [72](#)  
 importer (임포터), [72](#)  
 importing (임포트), [72](#)  
 interactive (대화형), [72](#)  
 interpreted (인터프리티드), [72](#)  
 interpreter shutdown (인터프리터 종료), [72](#)  
 iterable (이터러블), [72](#)  
 iterator (이터레이터), [73](#)

## K

key function (키 함수), [73](#)  
 keyword argument (키워드 인자), [73](#)

## L

lambda (람다), [73](#)  
 LBYL, [73](#)  
 list (리스트), [73](#)  
 list comprehension (리스트 컴프리헨션), [73](#)  
 loader (로더), [73](#)

## M

magic  
     method, [73](#)  
 magic method (매직 메서드), [73](#)  
 mapping (매핑), [73](#)  
 meta path finder (메타 경로 파인더), [74](#)  
 metaclass (메타 클래스), [74](#)  
 method  
     magic, [73](#)  
     special, [77](#)  
 method (메서드), [74](#)  
 method resolution order (메서드 결정 순서), [74](#)  
 module (모듈), [74](#)  
 module spec (모듈 스펙), [74](#)  
 MRO, [74](#)  
 mutable (가변), [74](#)

## N

named tuple (네임드 튜플), [74](#)  
 namespace (이름 공간), [74](#)  
 namespace package (이름 공간 패키지), [74](#)  
 nested scope (중첩된 스코프), [75](#)  
 new-style class (뉴스타일 클래스), [75](#)

## O

object  
     deallocation, [48](#)  
     finalization, [48](#)  
 object (객체), [75](#)

## P

package (패키지), [75](#)  
 parameter (매개변수), [75](#)

path based finder (경로 기반 파인더), [76](#)  
 path entry (경로 엔트리), [75](#)  
 path entry finder (경로 엔트리 파인더), [75](#)  
 path entry hook (경로 엔트리 훅), [75](#)  
 path-like object (경로류 객체), [76](#)  
 PEP, [76](#)  
 Philbrick, Geoff, [15](#)  
 portion (포션), [76](#)  
 positional argument (위치 인자), [76](#)  
 provisional API (잠정 API), [76](#)  
 provisional package (잠정 패키지), [76](#)  
 PyArg\_ParseTuple(), [13](#)  
 PyArg\_ParseTupleAndKeywords(), [14](#)  
 PyErr\_Fetch(), [48](#)  
 PyErr\_Restore(), [48](#)  
 PyInit\_modulename (C 함수), [55](#)  
 PyObject\_CallObject(), [12](#)  
 Python 3000 (파이썬 3000), [76](#)  
 Pythonic (파이썬다운), [76](#)  
 PYTHONPATH, [55](#)

## Q

qualified name (정규화된 이름), [76](#)

## R

READ\_RESTRICTED, [51](#)  
 READONLY, [51](#)  
 reference count (참조 횟수), [77](#)  
 regular package (정규 패키지), [77](#)  
 repr  
     네장 함수, [49](#)  
 RESTRICTED, [51](#)

## S

sequence (시퀀스), [77](#)  
 set comprehension, [77](#)  
 single dispatch (싱글 디스패치), [77](#)  
 slice (슬라이스), [77](#)  
 special  
     method, [77](#)  
 special method (특수 메서드), [77](#)  
 statement (문장), [77](#)  
 string  
     object representation, [49](#)

## T

text encoding (텍스트 인코딩), [77](#)  
 text file (텍스트 파일), [77](#)  
 triple-quoted string (삼중 따옴표 된 문자열), [78](#)  
 type (형), [78](#)  
 type alias (형 에일리어스), [78](#)  
 type hint (형 힌트), [78](#)

## U

universal newlines (유니버설 줄 넘김), [78](#)

## V

variable annotation (변수 어노테이션), [78](#)  
 virtual environment (가상 환경), [78](#)  
 virtual machine (가상 기계), [79](#)

## W

WRITE\_RESTRICTED, [51](#)

## X

내장 함수  
 repr, [49](#)

## Y

파이썬 향상 제안

PEP 1, [76](#)  
 PEP 238, [71](#)  
 PEP 278, [78](#)  
 PEP 302, [71](#), [73](#)  
 PEP 343, [69](#)  
 PEP 362, [68](#), [75](#)  
 PEP 411, [76](#)  
 PEP 420, [71](#), [74](#), [76](#)  
 PEP 442, [48](#)  
 PEP 443, [71](#)  
 PEP 451, [71](#)  
 PEP 484, [67](#), [71](#), [78](#)  
 PEP 489, [11](#), [55](#)  
 PEP 492, [68](#), [69](#)  
 PEP 498, [70](#)  
 PEP 519, [76](#)  
 PEP 525, [68](#)  
 PEP 526, [67](#), [78](#)  
 PEP 3116, [78](#)  
 PEP 3155, [76](#)

환경 변수

PYTHONPATH, [55](#)

## Z

Zen of Python (파이썬 젠), [79](#)