The Python Language Reference

릴리스 *3.14.0a7*

Guido van Rossum and the Python development team

4월 27, 2025

Contents

이 참조 설명서는 언어의 문법과 "중심 개념들(core semantics)"을 설명합니다. 딱딱하더라도 정확하고 완전해지려고 합니다. 중심에서 벗어난 내장형, 내장 함수, 모듈들의 개념들은 library-index 에 기술되어 있습니다. 언어에 대한 비형식적인 소개는 tutorial-index 에서 제공됩니다. C와 C++ 프로그래머를 위해서는 두 개의 설명서가 따로 제공됩니다: extending-index 는 파이썬 확장 모듈을 작성하는 방법에 대한 큰 그림을 설명하고, c-api-index 은 C/C++ 프로그래머에게 제공되는 인터페이스들을 상세하게 기술합니다.

CHAPTER 1

개요

이 레퍼런스 설명서는 파이썬 프로그래밍 언어를 설명합니다. 자습서를 목표로 하고 있지 않습니다.

가능한 한 정확하려고 노력하고 있지만, 문법과 어휘 분석 이외의 모든 것에는 형식 규격보다는 자연어를 사용합니다. 이 선택이 평균적인 독자들이 문서를 좀 더 잘 이해하도록 만들지만, 동시에 모호해질 가능성 역시 만듭니다. 결과적으로, 만약 여러분이 화성에서 왔고 이 문서만으로 파이썬을 다시 구현하려고 하면, 아마도 여러 가지를 짐작해야 할 것이고 결국 많이 다른 언어를 만드는 것으로 끝날 것입니다. 반면에, 여러분이 파이썬을 사용하고 있고 언어의 특정 영역에 대한 정확한 규칙에 대해 궁금해하고 있다면 거의 확실히 이곳에서 답을 찾을 수 있습니다. 좀 더 형식화된 정의를 보고 싶다면, 아마도 여러분의 시간을 기부하는 편이 좋습니다 — 그렇지 않으면 클로닝 기계를 발명하거나 :-).

참조 문서에 너무 많은 구현 세부 사항을 넣는 것은 위험합니다. 구현은 변경될 것이고 같은 언어의 다른 구현도 좀 다른 방식으로 동작할 수 있습니다. 반면에 (대안 구현이 점차 지지도를 높여가고 있기는 하지만) CPython 은 가장 널리 사용되는 파이썬 구현이고, 그것의 특별한 경우 들은 때로 언급할 가치가 있습니다. 구현이 추가의 제약을 내포하고 있는 경우는 특히 그렇습니다. 그래서, 텍스트 중간중간 짧은 "구현 노트" 가 튀어나오는 것을 보게 될 것입니다.

모든 파이썬 구현에는 많은 내장 표준 모듈들이 따라옵니다. 이것들은 library-index 에 기술되어 있습니다. 언어 정의에 주목할 만한 방식으로 관계될 경우 몇몇 내장 모듈들은 따로 언급됩니다.

1.1 대안 구현들

눈에 띄게 널리 사용되는 파이썬 구현이 존재하기는 하지만, 특정한 관심사를 가진 대상들에게 호소력을 가진 여러 대안 구현들이 존재합니다.

알려진 구현들은:

CPython
: 원조이기도 하고 가장 잘 관리되고 있는 C로 작성된 파이썬 구현입니다. 언어의 새로운 기능은 보통 여기에서 처음 등장합니다.

Jython
: Python implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website.

Python for .NET
: 이 구현은 실제로는 CPython 구현을 사용하지만, 매니지드(managed) .NET 응용 프로그램이고 .NET 라이브러리를 제공합니다. Bryan Lloyd가 만들었습니다다. 더 자세한 정보는 Python for .NET 홈페이지 에서 제공됩니다.

IronPython
: An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For more information, see the IronPython website.

PyPy
: An implementation of Python written completely in Python. It supports several advanced features not found in other implementations like stackless support and a Just in Time compiler. One of the goals of the project is to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project' s home page.

각 구현은 이 설명서에서 설명되는 언어와 조금씩 각기 다른 방법으로 벗어나거나, 표준 파이썬 문서에서 다루는 범위 밖의 특별한 정보들을 소개합니다. 여러분이 사용 중인 구현에 대해 어떤 것을 더 알아야 하는지 판단하기 위해서는 구현 별로 제공되는 문서를 참조할 필요가 있습니다.

1.2 표기법

The descriptions of lexical analysis and syntax use a modified Backus–Naur form (BNF) grammar notation. This uses the following style of definition:

```
name:      lc_letter (lc_letter | "_")*
lc_letter: "a"..."z"
```

첫 줄은 `name` 이 `lc_letter` 로 시작하고, 없거나 하나 이상의 `lc_letter` 나 밑줄이 뒤따르는 형태로 구성된다고 말합니다. 한편 `lc_letter` 는 `'a'` 와 `'z'` 사이의 문자 하나입니다. (사실 이 규칙은 이 문서에서 어휘와 문법 규칙에서 정의되는 이름들에 대한 규칙입니다.)

개별 규칙은 이름 (위 규칙에 등장하는 `name`)과 `::=` 로 시작합니다. 세로막대(`|`)는 대안들을 분리하는 데 사용됩니다; 이 표기법에서 우선순위가 가장 낮은 연산자입니다. 별표(`*`)는 앞에 나오는 항목이 생략되거나 한 번 이상 반복될 수 있다는 의미입니다; 비슷하게, 더하기(`+`)는 한 번 이상 반복될 수 있지만 생략할 수는 없다는 뜻이고, 대괄호(`[]`)로 둘러싸인 것은 최대 한 번 나올 수 있고, 생략 가능하다는 뜻입니다. `*` 와 `+` 연산자는 최대한 엄격하게 연결됩니다; 우선순위가 가장 높습니다; 괄호는 덩어리로 묶는 데 사용됩니다. 문자열 리터럴은 따옴표로 둘러싸입니다. 공백은 토큰을 분리하는 용도로만 사용됩니다. 규칙은 보통 한 줄로 표현됩니다; 대안이 많은 규칙은 여러 줄로 표현될 수도 있는데, 뒤따르는 줄들이 세로막대로 시작되게 만듭니다.

어휘 정의 (위에서 든 예와 같이) 에서는, 두 가지 추가 관례가 사용됩니다: 두 개의 리터럴 문자가 세 개의 점으로 분리되어 있으면 주어진 (끝의 두 문자 모두 포함하는) 범위의 ASCII 문자 중 어느 하나라는 뜻입니다. 홑화살괄호(`<...>`) 안에 들어있는 구문은, 정의되는 기호에 대한 비형식적 설명을 제공합니다. 즉 필요한 경우 '제어 문자' 를 설명하는데 사용될 수 있습니다.

사용되는 표기법이 거의 같다고 하더라도, 어휘와 문법 정의 간에는 커다란 차이가 있습니다: 어휘 정의는 입력의 개별 문자에 적용되는 반면, 문법 정의는 어휘 분석기가 만들어내는 토큰들에 적용됩니다. 다음 장 ("어휘 분석(Lexical Analysis)") 에서 사용되는 모든 BNF는 어휘 정의입니다; 그 이후의 장에서는 문법 정의입니다.

CHAPTER 2

어휘 분석

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer* (also known as the *tokenizer*). This chapter describes how the lexical analyzer breaks a file into tokens.

파이썬은 프로그램 텍스트를 유니코드 코드값으로 읽습니다; 소스 파일의 인코딩은 인코딩 선언을 통해 지정될 수 있고, 기본값은 UTF-8 입니다. 자세한 내용은 **PEP 3120** 에 나옵니다. 소스 파일을 디코딩할 수 없을 때는 `SyntaxError` 가 발생합니다.

2.1 줄 구조(Line structure)

파이썬 프로그램은 여러 개의 논리적인 줄(*logical lines*) 들로 나뉩니다.

2.1.1 논리적인 줄

논리적인 줄의 끝은 NEWLINE 토큰으로 표현됩니다. 문법이 허락하지 않는 이상 (예를 들어 복합문에서 문장들 사이) 문장은 논리적인 줄 간의 경계를 가로지를 수 없습니다. 논리적인 줄은 명시적이거나 묵시적인 줄 결합(*line joining*) 규칙에 따라 하나 이상의 물리적인 줄(*physical lines*) 들로 구성됩니다.

2.1.2 물리적인 줄

물리적인 줄은 줄의 끝을 나타내는 시퀀스로 끝나는 문자들의 시퀀스입니다. 소스 파일과 문자열에는 플랫폼들의 표준 줄 종료 시퀀스들이 모두 사용될 수 있습니다 - ASCII LF (개행문자)를 사용하는 유닉스 형, ASCII 시퀀스 CR LF(캐리지 리턴 다음에 오는 개행 문자)를 사용하는 윈도우 형, ASCII CR(캐리지 리턴)을 사용하는 예전의 매킨토시 형. 이 형태들은 플랫폼의 종류와 관계없이 동등하게 사용할 수 있습니다. 입력의 끝은 마지막 물리적인 줄의 묵시적 종결자 역할을 합니다.

파이썬을 내장할 때는, 소스 코드 문자열은 반드시 줄 종료 문자에 표준 C 관행(ASCII LF를 표현하는 `\n` 문자로 줄이 종료됩니다)을 적용해서 파이썬 API로 전달되어야 합니다.

2.1.3 주석

주석은 문자열 리터럴에 포함되지 않는 해시 문자(#)로 시작하고 물리적인 줄의 끝에서 끝납니다. 묵시적인 줄 결합 규칙이 유효하지 않은 이상, 주석은 논리적인 줄을 종료시킵니다. 주석은 문법이 무시합니다.

2.1.4 인코딩 선언

파이썬 스크립트의 첫 번 째나 두 번째 줄에 있는 주석이 정규식 `coding[=:]\s*([-\w.]+)` 과 매치되면, 이 주석은 인코딩 선언으로 처리됩니다. 이 정규식의 첫 번째 그룹은 소스 코드 파일의 인코딩 이름을 지정합니다. 인코딩 선언은 줄 전체에 홀로 나와야 합니다. 만약 두 번째 줄이라면, 첫 번째 줄 역시 주석만 있어야 합니다. 인코딩 선언의 권장 형태는 두 개입니다. 하나는

```
# -*- coding: <encoding-name> -*-
```

인데 GNU Emacs에서도 인식됩니다. 다른 하나는

```
# vim:fileencoding=<encoding-name>
```

인데 Bram Moolenaar 의 VIM에서 인식됩니다.

If no encoding declaration is found, the default encoding is UTF-8. If the implicit or explicit encoding of a file is UTF-8, an initial UTF-8 byte-order mark (b' xefxbbxbf') is ignored rather than being a syntax error.

If an encoding is declared, the encoding name must be recognized by Python (see standard-encodings). The encoding is used for all lexical analysis, including string literals, comments and identifiers.

2.1.5 명시적인 줄 결합

둘 이상의 물리적인 줄은 역 슬래시 문자(\)를 사용해서 논리적인 줄로 결합할 수 있습니다: 물리적인 줄이 문자열 리터럴이나 주석의 일부가 아닌 역 슬래시 문자로 끝나면, 역 슬래시와 뒤따르는 개행 문자가 제거된 채로, 현재 만들어지고 있는 논리적인 줄에 합쳐집니다. 예를 들어:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

역 슬래시로 끝나는 줄은 주석이 포함될 수 없습니다. 역 슬래시는 주석을 결합하지 못합니다. 역 슬래시는 문자열 리터럴을 제외한 어떤 토큰도 결합하지 못합니다 (즉, 문자열 리터럴 이외의 어떤 토큰도 역 슬래시를 사용해서 두 줄에 나누어 기록할 수 없습니다.). 문자열 리터럴 밖에 있는 역 슬래시가 앞에서 언급한 장소 이외의 곳에 등장하는 것은 문법에 어긋납니다.

2.1.6 묵시적인 줄 결합

괄호(`()`), 대괄호(`[]`), 중괄호(`{}`)가 사용되는 표현은 역 슬래시 없이도 여러 개의 물리적인 줄로 나눌 수 있습니다. 예를 들어:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

묵시적으로 이어지는 줄들은 주석을 포함할 수 있습니다. 이어지는 줄들의 들여쓰기는 중요하지 않습니다. 중간에 빈 줄이 들어가도 됩니다. 묵시적으로 줄 결합하는 줄 들 간에는 NEWLINE 토큰이 만들어지지 않습니다. 묵시적으로 이어지는 줄들은 삼중 따옴표 된 문자열들에서도 등장할 수 있는데 (아래를 보라), 이 경우는 주석이 포함될 수 없습니다.

2.1.7 빈 줄

스페이스, 탭, 폼 피드(formfeed) 와 주석만으로 구성된 논리적인 줄은 무시됩니다. (즉 NEWLINE 토큰이 만들어지지 않습니다.) 대화형으로 문장이 입력되는 도중에는 빈 줄의 처리가 REPL 구현에 따라 달라질 수 있습니다. 표준 대화형 인터프리터에서는, 완전히 빈 줄(즉 공백이나 주석조차 없는 것)은 다중 행 문장을 종료시킵니다.

2.1.8 들여쓰기

논리적인 줄의 제일 앞에 오는 공백(스페이스와 탭)은 줄의 들여쓰기 수준을 계산하는 데 사용되고, 이는 다시 문장들의 묶음을 결정하는 데 사용되게 됩니다.

탭은 (왼쪽에서 오른쪽으로) 1~8개의 스페이스로 변환되는데, 치환된 후의 총 스페이스 문자 수가 8의 배수가 되도록 맞춥니다. (유닉스에서 사용되는 규칙에 맞추려는 것입니다.) 첫 번째 비 공백 문자 앞에 나오는 공백의 총수가 줄의 들여쓰기를 결정합니다. 들여쓰기는 역 슬래시를 사용해서 여러 개의 물리적인 줄로 나눠질 수 없습니다; 첫 번째 역 슬래시 이전의 공백이 들여쓰기를 결정합니다.

소스 파일이 탭과 스페이스를 섞어 쓰는 경우, 탭이 몇 개의 스페이스에 해당하는지에 따라 다르게 해석될 수 있으면 `TabError` 를 일으킵니다.

크로스-플랫폼 호환성 유의 사항: UNIX 이외의 플랫폼에서 편집기들이 동작하는 방식 때문에, 하나의 파일 내에서 들여쓰기를 위해 탭과 스페이스를 섞어 쓰는 것은 현명한 선택이 아닙니다. 다른 플랫폼들에서는 최대 들여쓰기 수준에 제한이 있을 수도 있다는 점도 주의해야 합니다.

폼 피드 문자는 줄의 처음에 나올 수 있습니다; 앞서 설명한 들여쓰기 수준 계산에서는 무시됩니다. 페이지 넘김 문자 앞에 공백이나 탭이 있는 경우는 정의되지 않은 효과를 줄 수 있습니다 (가령, 스페이스 수가 0으로 초기화될 수 있습니다).

연속된 줄의 들여쓰기 수준은, 스택을 사용해서, 다음과 같은 방법으로 INDENT와 DEDENT 토큰을 만드는 데 사용됩니다.

파일의 첫 줄을 읽기 전에 0하나를 스택에 넣습니다(push); 이 값은 다시 꺼내는(pop) 일이 없습니다. 스택에 넣는 값은 항상 스택의 아래에서 위로 올라갈 때 단조 증가합니다. 각 논리적인 줄의 처음에서 줄의 들여쓰기 수준이 스택의 가장 위에 있는 값과 비교됩니다. 같다면 아무런 일도 일어나지 않습니다. 더 크다면 그 값을 스택에 넣고 하나의 INDENT 토큰을 만듭니다. 더 작다면 이 값은 스택에 있는 값 중 하나여야만 합니다. 이 값보다 큰 모든 스택의 값들을 꺼내고(pop), 꺼낸 횟수만큼의 DEDENT 토큰을 만듭니다. 파일의 끝에서, 스택에 남아있는 0보다 큰 값의 개수만큼 DEDENT 토큰을 만듭니다.

여기에 (혼란스럽다 할지라도) 올바르게 들여쓰기 된 파이썬 코드 조각이 있습니다:

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

다음 예는 여러 가지 들여쓰기 에러를 보여줍니다:

```
 def perm(l):                       # error: first line indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

(사실, 처음 세 개의 에러는 파서가 감지합니다. 단지 마지막 에러만 어휘 분석기가 감지합니다. — `return r` 의 들여쓰기가 스택에 있는 값과 일치하지 않습니다.)

2.1.9 토큰 사이의 공백

논리적인 줄의 처음과 문자열 리터럴을 제외하고, 공백 문자인 스페이스, 탭, 폼 피드는 토큰을 분리하기 위해 섞어 쓸 수 있습니다. 두 토큰을 붙여 쓸 때 다른 토큰으로 해석될 수 있는 경우만 토큰 사이에 공백이 필요합니다. (예를 들어, ab 는 하나의 토큰이지만, a b 는 두 개의 토큰입니다.)

2.2 다른 토큰들

NEWLINE, INDENT, DEDENT 와는 별도로, 다음과 같은 유형의 토큰들이 존재합니다: 식별자(*identifier*), 키워드(*keyword*), 리터럴(*literal*), 연산자(*operator*), 구분자(*delimiter*). (앞에서 살펴본 줄 종료 이외의) 공백 문자들은 토큰이 아니지만, 토큰을 분리하는 역할을 담당합니다. 모호할 경우, 왼쪽에서 오른쪽으로 읽을 때, 하나의 토큰은 올바르고 가능한 한 최대 길이의 문자열로 구성되는 것을 선호합니다.

2.3 식별자와 키워드

식별자 (이름(*name*) 이라고도 합니다) 은 다음과 같은 어휘 정의로 기술됩니다.

파이썬에서 식별자의 문법은 유니코드 표준 부속서 UAX-31 에 기반을 두는데, 여기에 덧붙이거나 바꾼 내용은 아래에서 정의합니다. 좀 더 상세한 내용은 **PEP 3131** 에서 찾을 수 있습니다.

Within the ASCII range (U+0001..U+007F), the valid characters for identifiers include the uppercase and lowercase letters `A` through `Z`, the underscore _ and, except for the first character, the digits `0` through `9`. Python 3.0 introduced additional characters from outside the ASCII range (see **PEP 3131**). For these characters, the classification uses the version of the Unicode Character Database as included in the `unicodedata` module.

식별자는 길이에 제한이 없고, 케이스(case)는 구분됩니다.

```
identifier:   xid_start xid_continue*
id_start:     <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and
id_continue:  <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc and
xid_start:    <all characters in id_start whose NFKC normalization is in "id_start xid_continue*
xid_continue: <all characters in id_continue whose NFKC normalization is in "id_continue*">
```

위에서 언급한 유니코드 카테고리 코드들의 의미는 이렇습니다:

- *Lu* - uppercase letters
- *Ll* - lowercase letters
- *Lt* - titlecase letters
- *Lm* - modifier letters
- *Lo* - other letters
- *Nl* - letter numbers
- *Mn* - nonspacing marks
- *Mc* - spacing combining marks
- *Nd* - decimal numbers
- *Pc* - connector punctuations
- *Other_ID_Start* - explicit list of characters in PropList.txt to support backwards compatibility
- *Other_ID_Continue* - 마찬가지

모든 식별자는 파서에 의해 NFKC 정규화 형식으로 변환되고, 식별자의 비교는 NFKC 에 기반을 둡니다.

A non-normative HTML file listing all valid identifier characters for Unicode 16.0.0 can be found at https://www.unicode.org/Public/16.0.0/ucd/DerivedCoreProperties.txt

2.3.1 키워드

다음 식별자들은 예약어, 또는 언어의 키워드, 로 사용되고, 일반적인 식별자로 사용될 수 없습니다. 여기 쓰여 있는 것과 정확히 같게 사용되어야 합니다:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.3.2 Soft Keywords

Added in version 3.10.

Some identifiers are only reserved under specific contexts. These are known as *soft keywords*. The identifiers `match`, `case`, `type` and _ can syntactically act as keywords in certain contexts, but this distinction is done at the parser level, not when tokenizing.

As soft keywords, their use in the grammar is possible while still preserving compatibility with existing code that uses these names as identifier names.

`match`, `case`, and _ are used in the *`match`* statement. `type` is used in the *`type`* statement.

버전 3.12에서 변경: `type` is now a soft keyword.

2.3.3 식별자의 예약 영역

(키워드와는 별개로) 어떤 부류의 식별자들은 특별한 의미가 있습니다. 이 부류의 식별자들은 시작과 끝의 밑줄 문자 패턴으로 구분됩니다:

_*

Not imported by `from module import *`.

_

In a `case` pattern within a *`match`* statement, _ is a *soft keyword* that denotes a *wildcard*.

Separately, the interactive interpreter makes the result of the last evaluation available in the variable _. (It is stored in the `builtins` module, alongside built-in functions like `print`.)

Elsewhere, _ is a regular identifier. It is often used to name "special" items, but it is not special to Python itself.

> **참고**
>
> 이름 _ 은 종종 국제화(internationalization)와 관련되어 사용됩니다. 이 관례에 관해서는 `gettext` 모듈의 문서를 참조하십시오.
>
> It is also commonly used for unused variables.

__*__

시스템 정의 이름, 비공식적으로 "던더(dunder)" 이름이라고 알려졌습니다. 이 이름들은 인터프리터와 그 구현(표준 라이브러리를 포함합니다)이 정의합니다. 현재 정의된 시스템 이름은 특수 메서드 이름들 섹션과 그 외의 곳에서 논의됩니다. 파이썬의 미래 버전에서는 더 많은 것들이 정의될 가능성이 큽니다. 어떤 문맥에서건, 명시적으로 문서로 만들어진 사용법을 벗어나는 __*__ 이름의 모든 사용은, 경고 없이 손상될 수 있습니다.

__*

클래스-비공개 이름. 이 부류의 이름들을 클래스 정의 문맥에서 사용하면 뒤섞인 형태로 변형됩니다.

부모 클래스와 자식 클래스의 "비공개(private)" 어트리뷰트 간의 이름 충돌을 피하기 위함입니다. 식별자(이름) 섹션을 보세요.

2.4 리터럴

리터럴(literal)은 몇몇 내장형들의 상숫값을 위한 표기법입니다.

2.4.1 문자열과 바이트열 리터럴

문자열 리터럴은 다음과 같은 어휘 정의로 기술됩니다:

```
stringliteral:   [stringprefix](shortstring | longstring)
stringprefix:    "r" | "u" | "R" | "U" | "f" | "F"
                 | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring:     "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring:      "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem: shortstringchar | stringescapeseq
longstringitem:  longstringchar | stringescapeseq
shortstringchar: <any source character except "\" or newline or the quote>
longstringchar:  <any source character except "\">
stringescapeseq: "\" <any source character>

bytesliteral:   bytesprefix(shortbytes | longbytes)
bytesprefix:    "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes:     "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
longbytes:      "'''" longbytesitem* "'''" | '"""' longbytesitem* '"""'
shortbytesitem: shortbyteschar | bytesescapeseq
longbytesitem:  longbyteschar | bytesescapeseq
shortbyteschar: <any ASCII character except "\" or newline or the quote>
longbyteschar:  <any ASCII character except "\">
bytesescapeseq: "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the *stringprefix* or *bytesprefix* and the rest of the literal. The source character set is defined by the encoding declaration; it is UTF-8 if no encoding declaration is given in the source file; see section 인코딩 선언.

In plain English: Both types of literals can be enclosed in matching single quotes (`'`) or double quotes (`"`). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (`\`) character is used to give special meaning to otherwise ordinary characters like `n`, which means 'newline' when escaped (`\n`). It can also be used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. See *escape sequences* below for examples.

바이트열(bytes) 리터럴은 항상 `'b'` 나 `'B'` 를 앞에 붙입니다; `str` 형의 인스턴스 대신 `bytes` 형의 인스턴스를 만듭니다. 오직 ASCII 문자들만 포함할 수 있습니다. 코드값이 128보다 크거나 같은 값들은 반드시 이스케이핑으로 표현되어야 합니다.

Both string and bytes literals may optionally be prefixed with a letter `'r'` or `'R'`; such constructs are called *raw string literals* and *raw bytes literals* respectively and treat backslashes as literal characters. As a result, in raw string literals, `'\U'` and `'\u'` escapes are not treated specially.

Added in version 3.3: 날 바이트열 리터럴의 `'br'` 와 같은 의미가 있는 `'rb'` 접두어가 추가되었습니다.

파이썬 2.x 와 3.x 에서 동시에 지원하는 코드들의 유지보수를 단순화하기 위해 예전에 사용되던 유니코드 리터럴 (`u'value'`)이 다시 도입되었습니다. 자세한 정보는 **PEP 414** 에 나옵니다.

`'f'` 나 `'F'` 를 접두어로 갖는 문자열 리터럴은 포맷 문자열 리터럴(*formatted string literal*)입니다; *f-strings* 을 보세요. `'f'` 는 `'r'` 과 결합할 수 있습니다, 하지만, `'b'` 나 `'u'` 와는 결합할 수 없습니다. 따라서 날 포맷 문자열은 가능하지만, 포맷 바이트열 리터럴은 불가능합니다.

삼중 따옴표 된 리터럴에서, 세 개의 이스케이핑 되지 않은 개행 문자와 따옴표가 허락됩니다 (그리고 유지됩니다). 예외는 한 줄에 세 개의 이스케이핑 되지 않은 따옴표가 나오는 것인데, 리터럴을 종료시킵니다. ("따옴표"는 리터럴을 시작하는데 사용한 문자입니다. 즉, `'` 나 `"`)

Escape sequences

'r' 나 'R' 접두어가 붙지 않은 이상, 문자열과 바이트열 리터럴에 포함된 이스케이프 시퀀스는 표준 C에서 사용된 것과 비슷한 규칙으로 해석됩니다. 인식되는 이스케이프 시퀀스는 이렇습니다:

이스케이프 시퀀스	의미	유의 사항
\<newline>	역 슬래시와 개행 문자가 무시됩니다	(1)
\\	역 슬래시 (\)	
\'	작은따옴표 (')	
\"	큰따옴표 (")	
\a	ASCII 벨 (BEL)	
\b	ASCII 백스페이스 (BS)	
\f	ASCII 폼 피드 (FF)	
\n	ASCII 라인 피드 (LF)	
\r	ASCII 캐리지 리턴 (CR)	
\t	ASCII 가로 탭 (TAB)	
\v	ASCII 세로 탭 (VT)	
ooo	8진수 *ooo* 로 지정된 문자	(2,4)
\x*hh*	16진수 *hh* 로 지정된 문자	(3,4)

문자열 리터럴에서만 인식되는 이스케이프 시퀀스는:

이스케이프 시퀀스	의미	유의 사항
\N{*name*}	유니코드 데이터베이스에서 *name* 이라고 이름 붙여진 문자	(5)
\u*xxxx*	16-bit 16진수 *xxxx* 로 지정된 문자	(6)
\U*xxxxxxxx*	32-bit 16진수 *xxxxxxx* 로 지정된 문자	(7)

유의 사항:

(1) A backslash can be added at the end of a line to ignore the newline:

```
>>> 'This string will not include \
... backslashes or newline characters.'
'This string will not include backslashes or newline characters.'
```

The same result can be achieved using *triple-quoted strings*, or parentheses and *string literal concatenation*.

(2) 표준 C와 마찬가지로, 최대 세 개의 8진수가 허용됩니다.

버전 3.11에서 변경: Octal escapes with value larger than `0o377` produce a `DeprecationWarning`.

버전 3.12에서 변경: Octal escapes with value larger than `0o377` produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

(3) 표준 C와는 달리, 정확히 두 개의 16진수가 제공되어야 합니다.

(4) 바이트열 리터럴에서, 16진수와 8진수 이스케이프는 지정된 값의 바이트를 표현합니다. 문자열 리터럴에서는, 이 이스케이프는 지정된 값의 유니코드 문자를 표현합니다.

(5) 버전 3.3에서 변경: 별칭[1] 지원이 추가되었습니다

(6) 정확히 4개의 16진수를 필요로 합니다.

(7) 이 방법으로 모든 유니코드를 인코딩할 수 있습니다. 정확히 8개의 16진수가 필요합니다.

표준 C와는 달리, 인식되지 않는 모든 이스케이프 시퀀스는 문자열에 변경되지 않은 상태로 남게 됩니다. 즉, 역 슬래시가 결과에 남게 됩니다. (이 동작은 디버깅할 때 쓸모가 있습니다. 이스케이프 시퀀스가 잘못 입력되었을 때, 최종 결과에서 잘못된 부분을 쉽게 인지할 수 있습니다.) 문자열 리터럴에서만 인식되는 이스케이프 시퀀스가, 바이트열 리터럴에서는 인식되지 않는 부류임에 주목하십시오.

[1] https://www.unicode.org/Public/16.0.0/ucd/NameAliases.txt

버전 3.6에서 변경: Unrecognized escape sequences produce a `DeprecationWarning`.

버전 3.12에서 변경: Unrecognized escape sequences produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

날 리터럴에서 조차, 따옴표는 역 슬래시로 이스케이프 됩니다. 하지만 역 슬래시가 결과에 남게 됩니다; 예를 들어, `r"\""` 는 올바른 문자열 리터럴인데, 두 개의 문자가 들어있습니다: 역 슬래시와 큰따옴표; `r"\"` 는 올바른 문자열 리터럴이 아닙니다 (날 문자열조차 홀수개의 역 슬래시로 끝날 수 없습니다.). 좀 더 명확하게 말하자면, 날 리터럴은 하나의 역 슬래시로 끝날 수 없습니다(역 슬래시가 뒤에 오는 따옴표를 이스케이프 시키기 때문입니다). 역 슬래시와 바로 뒤에 오는 개행문자는 줄 결합이 아니라 리터럴에 포함되는 두 개의 문자로 인식됨에 주의해야 합니다.

2.4.2 문자열 리터럴 이어붙이기

여러 개의 문자열이나 바이트열 리터럴을 (공백으로 분리해서) 여러 개 인접해서 나열하는 것이 허락되고, 그 의미는 이어붙인 것과 같습니다. 각 리터럴이 서로 다른 따옴표를 사용해도 됩니다. 그래서, `"hello" 'world'` 는 `"helloworld"` 와 동등합니다. 이 기능은 긴 문자열을 편의상 여러 줄로 나눌 때 필요한 역 슬래시를 줄여줍니다. 각 문자열 단위마다 주석을 붙이는 것도 가능합니다. 예를 들어:

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

이 기능이 문법 수준에서 정의되고는 있지만, 컴파일 시점에 구현됨에 주의해야 합니다. 실행 시간에 문자열 표현을 이어붙이기 위해서는 '+' 연산자를 사용해야 합니다. 리터럴 이어붙이기가 요소별로 다른 따옴표를 사용할 수 있고 (날 문자열과 삼중 따옴표 문자열을 이어붙이는 것조차 가능합니다), 포맷 문자열 리터럴을 보통 문자열 리터럴과 이어붙일 수 있음에 유의해야 합니다.

2.4.3 f-strings

Added in version 3.6.

포맷 문자열 리터럴(*formatted string literal*) 또는 *f*-문자열 (*f-string*) 은 `'f'` 나 `'F'` 를 앞에 붙인 문자열 리터럴입니다. 이 문자열은 치환 필드를 포함할 수 있는데, 중괄호 `{}` 로 구분되는 표현식입니다. 다른 문자열 리터럴이 항상 상숫값을 갖지만, 포맷 문자열 리터럴은 실행시간에 계산되는 표현식입니다.

이스케이프 시퀀스는 일반 문자열 리터럴처럼 디코딩됩니다 (동시에 날 문자열인 경우는 예외입니다). 디코딩 후에 문자열의 내용은 다음과 같은 문법을 따릅니다:

```
f_string:          (literal_char | "{{" | "}}" | replacement_field)*
replacement_field: "{" f_expression ["="] ["!" conversion] [":" format_spec] "}"
f_expression:      (conditional_expression | "*" or_expr)
                     ("," conditional_expression | "," "*" or_expr)* [","]
                   | yield_expression
conversion:        "s" | "r" | "a"
format_spec:       (literal_char | replacement_field)*
literal_char:      <any code point except "{", "}" or NULL>
```

중괄호 바깥 부분은 일반 리터럴처럼 취급되는데, 이중 중괄호 `'{{'` 나 `'}}'` 가 대응하는 단일 중괄호로 치환된다는 점만 예외입니다. 하나의 여는 중괄호 `'{'` 는 치환 필드를 시작시키는데, 파이썬 표현식이 뒤따릅니다. 평가 후 표현식 텍스트와 해당 값을 모두 표시하려면 (디버깅에 유용합니다), 표현식 뒤에 등호 `'='`를 추가할 수 있습니다. 느낌표 `'!'` 로 시작하는, 변환(conversion) 필드가 뒤따를 수 있습니다. 포맷 지정자(format specifier)도 덧붙일 수 있는데, 콜론 `':'` 으로 시작합니다. 치환 필드는 닫는 중괄호 `'}'` 로 끝납니다.

Expressions in formatted string literals are treated like regular Python expressions surrounded by parentheses, with a few exceptions. An empty expression is not allowed, and both *lambda* and assignment expressions `:=` must be surrounded by explicit parentheses. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right. Replacement expressions can contain newlines in both single-quoted and triple-quoted f-strings and they can contain comments. Everything that comes after a `#` inside a replacement field is a comment (even closing braces and quotes). In that case, replacement fields must be closed in a different line.

```
>>> f"abc{a # This is a comment }"
... + 3}"
'abc5'
```

버전 3.7에서 변경: 파이썬 3.7 이전에는, 구현 문제로 인해 포맷 문자열 리터럴의 표현식에서 *await* 표현식과 *async for* 절을 포함하는 컴프리헨션은 유효하지 않았습니다.

버전 3.12에서 변경: Prior to Python 3.12, comments were not allowed inside f-string replacement fields.

등호 기호 `'='`이 제공되면, 출력에는 표현식 텍스트, `'='` 및 평가된 값이 포함됩니다. 여는 중괄호 `'{'` 뒤, 표현식 내, `'='` 뒤의 스페이스는 모두 출력에 유지됩니다. 기본적으로, `'='`은 포맷 지정자가 없는 한 표현식의 `repr()`을 제공합니다. 포맷이 지정되면 변환 `'!r'`이 선언되지 않는 한 기본적으로 표현식의 `str()`이 사용됩니다.

Added in version 3.8: 등호 기호 `'='`.

변환(conversion)이 지정되면, 표현식의 결과가 포매팅 전에 변환됩니다. 변환 `'!s'`는 결과에 `str()`을 호출하고, `'!r'`은 `repr()`을 호출하고, `'!a'`은 `ascii()`를 호출합니다.

The result is then formatted using the `format()` protocol. The format specifier is passed to the *__format__()* method of the expression or conversion result. An empty string is passed when the format specifier is omitted. The formatted result is then included in the final value of the whole string.

Top-level format specifiers may include nested replacement fields. These nested fields may include their own conversion fields and format specifiers, but may not include more deeply nested replacement fields. The format specifier mini-language is the same as that used by the `str.format()` method.

포맷 문자열 리터럴을 이어붙일 수는 있지만, 치환 필드가 여러 리터럴로 쪼개질 수는 없습니다.

포맷 문자열 리터럴의 예를 들면:

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

Reusing the outer f-string quoting type inside a replacement field is permitted:

```
>>> a = dict(x=2)
>>> f"abc {a["x"]} def"
'abc 2 def'
```

버전 3.12에서 변경: Prior to Python 3.12, reuse of the same quoting type of the outer f-string inside a replacement field was not possible.

Backslashes are also allowed in replacement fields and are evaluated the same way as in any other context:

```
>>> a = ["a", "b", "c"]
>>> print(f"List a contains:\n{"\n".join(a)}")
List a contains:
a
b
c
```

버전 3.12에서 변경: Prior to Python 3.12, backslashes were not permitted inside an f-string replacement field.

포맷 문자열 리터럴은 독스트링(docstring)으로 사용될 수 없습니다. 표현식이 전혀 없더라도 마찬가집니다.

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

포맷 문자열 리터럴 추가에 대한 제안은 **PEP 498** 을 참조하고, 관련된 포맷 문자열 메커니즘을 사용하는 `str.format()` 도 살펴보는 것이 좋습니다.

2.4.4 숫자 리터럴

There are three types of numeric literals: integers, floating-point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

숫자 리터럴이 부호를 포함하지 않는 것에 주의해야 합니다; `-1` 과 같은 구문은 일 항 연산자 '`-`' 과 리터럴 `1` 로 구성된 표현식입니다.

2.4.5 정수 리터럴

정수 리터럴은 다음과 같은 어휘 정의로 표현됩니다:

```
integer:      decinteger | bininteger | octinteger | hexinteger
decinteger:   nonzerodigit (["_"] digit)* | "0"+ (["_"] "0")*
bininteger:   "0" ("b" | "B") (["_"] bindigit)+
octinteger:   "0" ("o" | "O") (["_"] octdigit)+
hexinteger:   "0" ("x" | "X") (["_"] hexdigit)+
nonzerodigit: "1"..."9"
digit:        "0"..."9"
bindigit:     "0" | "1"
octdigit:     "0"..."7"
hexdigit:     digit | "a"..."f" | "A"..."F"
```

가용한 메모리에 저장될 수 있는지와는 별개로 정수 리터럴의 길이에 제한은 없습니다.

밑줄은 리터럴의 숫자 값을 결정할 때 고려되지 않습니다. 가독성을 높이기 위해 숫자들을 무리 지을 때 쓸모가 있습니다. 밑줄은 숫자 사이나 `0x` 와 같은 진수 지정자(base specifier) 다음에 나올 수 있는데, 한 번에 하나만 사용될 수 있습니다.

0 이 아닌 10진수가 0으로 시작할 수 없음에 주의해야 합니다. 3.0 버전 이전의 파이썬에서 사용한 C 스타일의 8진수 리터럴과 혼동되는 것을 막기 위함입니다.

정수 리터럴의 예를 들면:

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```

버전 3.6에서 변경: 리터럴에서 숫자들의 그룹을 표현할 목적으로 밑줄을 허락합니다.

2.4.6 Floating-point literals

Floating-point literals are described by the following lexical definitions:

```
floatnumber:   pointfloat | exponentfloat
pointfloat:    [digitpart] fraction | digitpart "."
exponentfloat: (digitpart | pointfloat) exponent
digitpart:     digit (["_"] digit)*
fraction:      "." digitpart
exponent:      ("e" | "E") ["+" | "-"] digitpart
```

Note that the integer and exponent parts are always interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating-point literals is implementation-dependent. As in integer literals, underscores are supported for digit grouping.

Some examples of floating-point literals:

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

버전 3.6에서 변경: 리터럴에서 숫자들의 그룹을 표현할 목적으로 밑줄을 허락합니다.

2.4.7 허수 리터럴

허수 리터럴은 다음과 같은 어휘 정의로 표현됩니다:

```
imagnumber: (floatnumber | digitpart) ("j" | "J")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating-point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating-point number to it, e.g., `(3+4j)`. Some examples of imaginary literals:

```
3.14j   10.j    10j     .001j   1e100j   3.14e-10j   3.14_15_93j
```

2.5 연산자

다음과 같은 토큰들은 연산자입니다:

```
+       -       *       **      /       //      %      @
<<      >>      &       |       ^       ~       :=
<       >       <=      >=      ==      !=
```

2.6 구분자

다음 토큰들은 문법에서 구분자(delimiter)로 기능합니다:

```
(       )       [       ]       {       }
,       :       !       .       ;       @       =
->      +=      -=      *=      /=      //=     %=
@=      &=      |=      ^=      >>=     <<=     **=
```

마침표는 실수와 허수 리터럴에서도 등장할 수 있습니다. 연속된 마침표 세 개는 생략부호 리터럴(ellipsis literal)이라는 특별한 의미가 있습니다. 목록 후반의 증분 대입 연산자(augmented assignment operator)들은 어휘적으로는 구분자로 기능하지만, 동시에 연산을 수행합니다.

다음의 인쇄되는 ASCII 문자들은 다른 토큰들 일부로서 특별한 의미가 있거나, 그렇지 않으면 어휘 분석기에 유의미합니다:

```
'       "       #       \
```

다음의 인쇄되는 ASCII 문자들은 파이썬에서 사용되지 않습니다. 문자열 리터럴과 주석 이외의 곳에서 사용되는 것은 조건 없는 에러입니다:

```
$       ?       `
```

CHAPTER 3

데이터 모델

3.1 객체, 값, 형

객체 (*Objects*)는 파이썬이 데이터(data)를 추상화한 것(abstraction)입니다. 파이썬 프로그램의 모든 데이터는 객체나 객체 간의 관계로 표현됩니다. (폰 노이만(Von Neumann)의 "프로그램 내장식 컴퓨터(stored program computer)" 모델을 따르고, 또 그 관점에서 코드 역시 객체로 표현됩니다.)

Every object has an identity, a type and a value. An object' s *identity* never changes once it has been created; you may think of it as the object' s address in memory. The `is` operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

CPython 의 경우, `id(x)` 는 `x` 가 저장된 메모리의 주소입니다.

객체의 형은 객체가 지원하는 연산들을 정의하고 (예를 들어, "길이를 갖고 있나?") 그 형의 객체들이 가질 수 있는 가능한 값들을 정의합니다. `type()` 함수는 객체의 형(이것 역시 객체다)을 돌려줍니다. 아이덴티티와 마찬가지로, 객체의 형 (*type*) 역시 변경되지 않습니다.[1]

어떤 객체들의 값은 변경할 수 있습니다. 값을 변경할 수 있는 객체들을 가변(*mutable*) 이라고 합니다. 일단 만들어진 후에 값을 변경할 수 없는 객체들을 불변(*immutable*) 이라고 합니다. (가변 객체에 대한 참조를 저장하고 있는 불변 컨테이너의 값은 가변 객체의 값이 변할 때 변경된다고 볼 수도 있습니다; 하지만 저장하고 있는 객체들의 집합이 바뀔 수 없으므로 컨테이너는 여전히 불변이라고 여겨집니다. 따라서 불변성은 엄밀하게는 변경 불가능한 값을 갖는 것과는 다릅니다. 좀 더 미묘합니다.) 객체의 가변성(mutability)은 그것의 형에 의해 결정됩니다; 예를 들어 숫자, 문자열, 튜플(tuple)은 불변이지만, 딕셔너리(dictionary) 와 리스트(list)는 가변입니다.

객체는 결코 명시적으로 파괴되지 않습니다; 더 참조되지 않을 때(unreachable) 가비지 수거(garbage collect)됩니다. 구현이 가비지 수거를 지연시키거나 아예 생략하는 것이 허락됩니다 — 아직 참조되는 객체들을 수거하지 않는 이상 가비지 수거가 어떤 식으로 구현되는지는 구현의 품질 문제입니다.

CPython 은 현재 참조 횟수 계산(reference-counting) 방식을 사용하는데, (선택 사항으로) 순환적으로 연결된 가비지의 지연된 감지가 추가됩니다. 이 방법으로 대부분 객체를 참조가 제거되자마자 수거할 수 있습니다. 하지만 순환 참조가 있는 가비지들을 수거한다는 보장은 없습니다. 순환적 가비지 수거의 제어에 관한 정보는 `gc` 모듈 문서를 참조하면 됩니다. 다른 구현들은 다른 식으로 동작하고, CPython 도 변경될 수 있습니다. 참조가 제거될 때 즉각적으로 파이널리제이션(finalization)되는 것에 의존하지 말아야 합니다 (그래서 항상 파일을 명시적으로 닫아주어야 합니다).

Note that the use of the implementation' s tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a `try`…`except` statement may keep objects alive.

[1] 어떤 제한된 조건으로, 어떤 경우에 객체의 형을 변경하는 것이 가능합니다. 하지만 잘못 다뤄지면 아주 괴상한 결과로 이어질 수 있으므로 일반적으로 좋은 생각이 아닙니다.

Some objects contain references to "external" resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The *try*⋯*finally* statement and the *with* statement provide convenient ways to do this.

어떤 객체들은 다른 객체에 대한 참조를 포함하고 있습니다. 이런 것들을 컨테이너(*container*) 라고 부릅니다. 튜플, 리스트, 딕셔너리등이 컨테이너의 예입니다. 이 참조들은 컨테이너의 값의 일부입니다. 대부분은, 우리가 컨테이너의 값을 논할 때는, 들어있는 객체들의 아이덴티티 보다는 값을 따집니다. 하지만, 컨테이너의 가변성에 대해 논할 때는 직접 가진 객체들의 아이덴티티만을 따집니다. 그래서, (튜플 같은) 불변 컨테이너가 가변 객체로의 참조를 하고 있다면, 그 가변 객체가 변경되면 컨테이너의 값도 변경됩니다.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. For example, after `a = 1; b = 1`, *a* and *b* may or may not refer to the same object with the value one, depending on the implementation. This is because `int` is an immutable type, so the reference to `1` can be reused. This behaviour depends on the implementation used, so should not be relied upon, but is something to be aware of when making use of object identity tests. However, after `c = []; d = []`, *c* and *d* are guaranteed to refer to two different, unique, newly created empty lists. (Note that `e = f = []` assigns the *same* object to both *e* and *f*.)

3.2 표준형 계층

아래에 파이썬에 내장된 형들의 목록이 있습니다. (구현에 따라 C 나 자바나 다른 언어로 작성된) 확장 모듈들은 추가의 형을 정의할 수 있습니다. 파이썬의 미래 버전 역시 형 계층에 형을 더할 수 있는데 (예를 들어, 유리수, 효율적으로 저장된 정수 배열 등등), 표준 라이브러리를 통해 추가될 가능성이 더 크기는 합니다.

아래에 나오는 몇몇 형에 대한 설명은 '특수 어트리뷰트(special attribute)' 를 나열하는 문단을 포함합니다. 이것들은 구현에 접근할 방법을 제공하는데, 일반적인 사용을 위한 것이 아닙니다. 정의는 앞으로 변경될 수 있습니다.

3.2.1 None

이 형은 하나의 값만을 갖습니다. 이 값을 갖는 하나의 객체가 존재합니다. 이 객체에는 내장된 이름 `None` 을 통해 접근합니다. 여러 가지 상황에서 값의 부재를 알리는 데 사용됩니다. 예를 들어, 명시적으로 뭔가를 돌려주지 않는 함수의 반환 값입니다. 논리값은 거짓입니다.

3.2.2 NotImplemented

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods should return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) It should not be evaluated in a boolean context.

더 자세한 내용은 implementing-the-arithmetic-operations 을 참고하십시오.

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

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

3.2.3 Ellipsis

이 형은 하나의 값만을 갖습니다. 이 값을 갖는 하나의 객체가 존재합니다. 이 객체에는 리터럴 `...` 이나 내장된 이름 `Ellipsis` 을 통해 접근합니다. 논리값은 참입니다.

3.2.4 numbers.Number

이것들은 숫자 리터럴에 의해 만들어지고, 산술 연산과 내장 산술 함수들이 결과로 돌려줍니다. 숫자 객체는 불변입니다; 한 번 값이 만들어지면 절대 변하지 않습니다. 파이썬의 숫자는 당연히 수학적인 숫자들과 밀접하게 관련되어 있습니다, 하지만 컴퓨터의 숫자 표현상의 제약을 받고 있습니다.

The string representations of the numeric classes, computed by *__repr__()* and *__str__()*, have the following properties:

- 클래스 생성자에 전달될 때 원래 숫자 값을 가진 객체를 생성하는 유효한 숫자 리터럴 입니다.
- 가능하면, 표현은 10진법입니다.
- 소수점 앞의 단일 0을 제외하고, 선행 0은 표시되지 않습니다.
- 소수점 뒤의 단일 0을 제외하고, 후행 0은 표시되지 않습니다.
- 부호는 숫자가 음수일 때만 표시됩니다.

Python distinguishes between integers, floating-point numbers, and complex numbers:

numbers.Integral

이것들은 수학적인 정수 집합(양과 음)에 속하는 요소들을 나타냅니다.

> **참고**
>
> 정수 표현 규칙은 음수가 포함된 시프트와 마스크 연산에 가장 의미 있는 해석을 제공하기 위한 것입니다.

두 가지 종류의 정수가 있습니다:

정수 (int)
: 이것은 (가상) 메모리가 허락하는 한, 제약 없는 범위의 숫자를 표현합니다. 시프트(shift)와 마스크(mask) 연산이 목적일 때는 이진 표현이 가정되고, 음수는 일종의 2의 보수(2' s complement)로 표현되는데, 부호 비트가 왼쪽으로 무한히 확장된 것과 같은 효과를 줍니다.

불린 (bool)
: 이것은 논리값 거짓과 참을 나타냅니다. `False` 와 `True` 두 객체만 불린 형 객체입니다. 불린 형은 int 형의 자식형(subtype)이고, 대부분 상황에서 각기 0과1처럼 동작합니다. 예외는 문자열로 변환되는 경우인데, 각기 문자열 `"False"` 와 `"True"` 가 반환됩니다.

numbers.Real (float)

These represent machine-level double precision floating-point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating-point numbers; the savings in processor and memory usage that are usually the reason for using these are dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating-point numbers.

numbers.Complex (complex)

These represent complex numbers as a pair of machine-level double precision floating-point numbers. The same caveats apply as for floating-point numbers. The real and imaginary parts of a complex number `z` can be retrieved through the read-only attributes `z.real` and `z.imag`.

3.2.5 시퀀스들

These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is *n*, the index set contains the numbers 0, 1, ⋯, *n*-1. Item *i* of sequence *a* is selected by `a[i]`. Some sequences, including built-in sequences, interpret negative subscripts by adding the sequence length. For example, `a[-2]` equals `a[n-2]`, the second to last item of sequence a with length `n`.

Sequences also support slicing: `a[i:j]` selects all items with index *k* such that *i* <= *k* < *j*. When used as an expression, a slice is a sequence of the same type. The comment above about negative indexes also applies to negative slice positions.

어떤 시퀀스는 세 번째 "스텝(step)" 매개변수를 사용하는 "확장 슬라이싱(extended slicing)"도 지원합니다: `a[i:j:k]` 는 `x = i + n*k`, *n* `>= 0`, *i* <= *x* < *j* 를 만족하는 모든 항목 *x* 를 선택합니다.

시퀀스는 불변성에 따라 구분됩니다

불변 시퀀스

불변 시퀀스 형의 객체는 일단 만들어진 후에는 변경될 수 없습니다. (만약 다른 객체로의 참조를 포함하면, 그 객체는 가변일 수 있고, 변경될 수 있습니다; 하지만, 불변 객체로부터 참조되는 객체의 집합 자체는 변경될 수 없습니다.)

다음과 같은 형들은 불변 시퀀스입니다:

문자열(Strings)
: A string is a sequence of values that represent Unicode code points. All the code points in the range `U+0000 - U+10FFFF` can be represented in a string. Python doesn' t have a `char` type; instead, every code point in the string is represented as a string object with length `1`. The built-in function `ord()` converts a code point from its string form to an integer in the range `0 - 10FFFF`; `chr()` converts an integer in the range `0 - 10FFFF` to the corresponding length `1` string object. `str.encode()` can be used to convert a `str` to `bytes` using the given text encoding, and `bytes.decode()` can be used to achieve the opposite.

튜플(Tuples)
: 튜플의 항목은 임의의 파이썬 객체입니다. 두 개 이상의 항목으로 구성되는 튜플은 콤마로 분리된 표현식의 목록으로 만들 수 있습니다. 하나의 항목으로 구성된 튜플(싱글턴,singleton)은 표현식에 콤마를 붙여서 만들 수 있습니다(괄호로 표현식을 묶을 수 있으므로, 표현식 만으로는 튜플을 만들지 않습니다). 빈 튜플은 한 쌍의 빈 괄호로 만들 수 있습니다.

바이트열(Bytes)
: A bytes object is an immutable array. The items are 8-bit bytes, represented by integers in the range 0 <= x < 256. Bytes literals (like `b'abc'`) and the built-in `bytes()` constructor can be used to create bytes objects. Also, bytes objects can be decoded to strings via the `decode()` method.

가변 시퀀스

가변 시퀀스는 만들어진 후에 변경될 수 있습니다. 서브스크립션(subscription)과 슬라이싱은 대입문과 *del* (삭제) 문의 대상으로 사용될 수 있습니다.

> **참고**
>
> The `collections` and `array` module provide additional examples of mutable sequence types.

현재 두 개의 내장 가변 시퀀스형이 있습니다:

리스트(Lists)
: 리스트의 항목은 임의의 파이썬 객체입니다. 리스트는 콤마로 분리된 표현식을 대괄호 안에 넣어서 만들 수 있습니다. (길이 0이나 1의 리스트를 만드는데 별도의 규칙이 필요 없습니다.)

바이트 배열(Byte Arrays)
: 바이트 배열(bytearray) 객체는 가변 배열입니다. 내장 `bytearray()` 생성자로 만들어집니다. 가변이라는 것(그래서 해싱 불가능하다는 것)을 제외하고, 바이트 배열은 불변 바이트열(`bytes`) 객체와 같은 인터페이스와 기능을 제공합니다.

3.2.6 집합 형들(Set types)

이것들은 중복 없는 불변 객체들의 순서 없고 유한한 집합을 나타냅니다. 인덱싱할 수 없습니다. 하지만 이터레이트할 수 있고, 내장 함수 `len()` 은 집합 안에 있는 항목들의 개수를 돌려줍니다. 집합의 일반적인 용도는 빠른 멤버십 검사(fast membership testing), 시퀀스에서 중복된 항목 제거, 교집합(intersection), 합집합(union), 차집합(difference), 대칭차집합(symmetric difference)과 같은 집합 연산을 계산하는 것입니다.

집합의 원소들에는 딕셔너리 키와 같은 불변성 규칙이 적용됩니다. 숫자 형의 경우는 숫자 비교에 관한 일반 원칙이 적용된다는 점에 주의해야 합니다: 만약 두 숫자가 같다고 비교되면(예를 들어, `1` 과 `1.0`), 그중 하나만 집합에 들어갈 수 있습니다.

현재 두 개의 내장 집합 형이 있습니다:

집합(Sets)
: 이것들은 가변 집합을 나타냅니다. 내장 `set()` 생성자로 만들 수 있고, `add()` 같은 메서드들을 사용해서 나중에 수정할 수 있습니다.

불변 집합(Frozen sets)
: 이것들은 불변 집합을 나타냅니다. 내장 `frozenset()` 생성자로 만들 수 있습니다. 불변 집합(frozenset)은 불변이고 해시 가능 하므로, 다른 집합의 원소나, 딕셔너리의 키로 사용될 수 있습니다.

3.2.7 매핑(Mappings)

이것들은 임의의 인덱스 집합으로 인덱싱되는 객체들의 유한한 집합을 나타냅니다. 인덱스 표기법(subscript notation) `a[k]` 는 매핑 `a` 에서 `k` 로 인덱스 되는 항목을 선택합니다; 이것은 표현식에 사용될 수도 있고, 대입이나 *del* 문장의 대상이 될 수도 있습니다. 내장 함수 `len()` 은 매핑에 포함된 항목들의 개수를 돌려줍니다.

현재 한 개의 내장 매핑 형이 있습니다:

딕셔너리(Dictionaries)

이것들은 거의 임의의 인덱스 집합으로 인덱싱되는 객체들의 유한한 집합을 나타냅니다. 키로 사용할 수 없는 것들은 리스트, 딕셔너리나 그 외의 가변형 중에서 아이덴티티가 아니라 값으로 비교되는 것들뿐입니다. 딕셔너리의 효율적인 구현이, 키의 해시값이 도중에 변경되지 않고 계속 같은 값으로 유지되도록 요구하고 있기 때문입니다. 키로 사용되는 숫자 형의 경우는 숫자 비교에 관한 일반 원칙이 적용됩니다: 만약 두 숫자가 같다고 비교되면(예를 들어, `1` 과 `1.0`), 둘 다 같은 딕셔너리 항목을 인덱싱하는데 사용될 수 있습니다.

딕셔너리는 삽입 순서를 유지합니다, 키가 딕셔너리에 순차적으로 추가된 순서와 같은 순서로 생성됨을 뜻합니다. 기존 키를 교체해도 순서는 변경되지 않지만, 키를 제거했다가 다시 삽입하면 이전 위치를 유지하는 대신 끝에 추가됩니다.

Dictionaries are mutable; they can be created by the `{}` notation (see section 딕셔너리 디스플레이).

확장 모듈 `dbm.ndbm` 과 `dbm.gnu` 는 추가의 매핑 형을 제공하는데, `collections` 모듈 역시 마찬가지입니다.

버전 3.7에서 변경: 딕셔너리는 3.6 이전의 파이썬 버전에서 삽입 순서를 유지하지 않았습니다. CPython 3.6에서, 삽입 순서가 유지되었지만, 그 시점에는 언어 보증이 아니라 구현 세부 사항으로 간주하였습니다.

3.2.8 콜러블(Callable types)

이것들은 함수 호출 연산(호출 섹션 참고)이 적용될 수 있는 형들입니다:

사용자 정의 함수

사용자 정의 함수 객체는 함수 정의를 통해 만들어집니다 (함수 정의 섹션 참고). 함수의 형식 매개변수(formal parameter) 목록과 같은 개수의 항목을 포함하는 인자(argument) 목록으로 호출되어야 합니다.

Special read-only attributes

어트리뷰트	의미
function.**__globals__**	A reference to the dictionary that holds the function's *global variables* – the global namespace of the module in which the function was defined.
function.**__closure__**	None or a tuple of cells that contain bindings for the names specified in the *co_freevars* attribute of the function's *code object*. 셀 객체는 cell_contents 어트리뷰트를 가지고 있습니다. 셀의 값을 읽을 뿐만 아니라 값을 설정하는 데도 사용할 수 있습니다.

Special writable attributes

Most of these attributes check the type of the assigned value:

어트리뷰트	의미
function.**__doc__**	The function's documentation string, or None if unavailable.
function.**__name__**	The function's name. See also: __name__ attributes.
function.**__qualname__**	The function's *qualified name*. See also: __qualname__ attributes. Added in version 3.3.
function.**__module__**	함수가 정의된 모듈의 이름 또는 (없는 경우) None
function.**__defaults__**	A tuple containing default *parameter* values for those parameters that have defaults, or None if no parameters have a default value.
function.**__code__**	The *code object* representing the compiled function body.
function.**__dict__**	The namespace supporting arbitrary function attributes. See also: *__dict__ attributes*.
function.**__annotations__**	A dictionary containing annotations of *parameters*. The keys of the dictionary are the parameter names, and 'return' for the return annotation, if provided. See also: *object.__annotations__*. 버전 3.14에서 변경: Annotations are now *lazily evaluated*. See **PEP 649**.
function.**__annotate__**	The *annotate function* for this function, or None if the function has no annotations. See *object.__annotate__*. Added in version 3.14.
function.**__kwdefaults__**	A dictionary containing defaults for keyword-only *parameters*.
function.**__type_params__**	A tuple containing the *type parameters* of a *generic function*. Added in version 3.12.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes.

CPython 구현 상세: CPython' s current implementation only supports function attributes on user-defined functions. Function attributes on *built-in functions* may be supported in the future.

Additional information about a function' s definition can be retrieved from its *code object* (accessible via the *__code__* attribute).

인스턴스 메서드(Instance methods)

인스턴스 메서드는 클래스, 클래스 인스턴스와 모든 콜러블 객체 (보통 사용자 정의 함수)을 결합합니다.

Special read-only attributes:

`method.__self__`	Refers to the class instance object to which the method is *bound*
`method.__func__`	Refers to the original *function object*
`method.__doc__`	The method' s documentation (same as *method.__func__.__doc__*). A `string` if the original function had a docstring, else `None`.
`method.__name__`	The name of the method (same as *method.__func__.__name__*)
`method.__module__`	The name of the module the method was defined in, or `None` if unavailable.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying *function object*.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined *function object* or a `classmethod` object.

When an instance method object is created by retrieving a user-defined *function object* from a class via one of its instances, its *__self__* attribute is the instance, and the method object is said to be *bound*. The new method' s *__func__* attribute is the original function object.

When an instance method object is created by retrieving a `classmethod` object from a class or instance, its *__self__* attribute is the class itself, and its *__func__* attribute is the function object underlying the class method.

When an instance method object is called, the underlying function (*__func__*) is called, inserting the class instance (*__self__*) in front of the argument list. For instance, when `C` is a class which contains a definition for a function `f()`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

When an instance method object is derived from a `classmethod` object, the "class instance" stored in *__self__* will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

It is important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

제너레이터 함수(Generator functions)

A function or method which uses the *yield* statement (see section *yield* 문) is called a *generator function*. Such a function, when called, always returns an *iterator* object which can be used to execute the body of the function: calling the iterator' s `iterator.__next__()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a *return* statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

코루틴 함수(Coroutine functions)

async def 를 사용해서 정의되는 함수나 메서드를 코루틴 함수 (*coroutine function*) 라고 부릅니다. 이런 함수를 호출하면 코루틴 객체를 돌려줍니다. *await* 표현식을 비롯해, *async with* 와 *async for* 문을 사용할 수 있습니다. 코루틴 객체 (*Coroutine Objects*) 섹션을 참조하십시오.

비동기 제너레이터 함수(Asynchronous generator functions)

A function or method which is defined using *async def* and which uses the *yield* statement is called a *asynchronous generator function*. Such a function, when called, returns an *asynchronous iterator* object which can be used in an *async for* statement to execute the body of the function.

Calling the asynchronous iterator' s *aiterator.__anext__* method will return an *awaitable* which when awaited will execute until it provides a value using the *yield* expression. When the function executes an empty *return* statement or falls off the end, a `StopAsyncIteration` exception is raised and the asynchronous iterator will have reached the end of the set of values to be yielded.

내장 함수(Built-in functions)

A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes:

- `__doc__` is the function' s documentation string, or `None` if unavailable. See *function.__doc__*.
- `__name__` is the function' s name. See *function.__name__*.
- `__self__` is set to `None` (but see the next item).
- `__module__` is the name of the module the function was defined in or `None` if unavailable. See *function.__module__*.

내장 메서드(Built-in methods)

This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming *alist* is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by *alist*. (The attribute has the same semantics as it does with *other instance methods*.)

클래스(Classes)

Classes are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override *__new__()*. The arguments of the call are passed to `__new__()` and, in the typical case, to *__init__()* to initialize the new instance.

클래스 인스턴스(Class Instances)

Instances of arbitrary classes can be made callable by defining a *__call__()* method in their class.

3.2.9 모듈(Modules)

Modules are a basic organizational unit of Python code, and are created by the *import system* as invoked either by the *import* statement, or by calling functions such as `importlib.import_module()` and built-in `__import__()`. A module object has a namespace implemented by a `dictionary` object (this is the dictionary referenced by the *__globals__* attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn' t needed once the initialization is done).

어트리뷰트 대입은 모듈의 이름 공간 딕셔너리를 갱신합니다. 예를 들어, `m.x = 1` 은 `m.__dict__["x"] = 1` 과 같습니다.

Import-related attributes on module objects

Module objects have the following attributes that relate to the *import system*. When a module is created using the machinery associated with the import system, these attributes are filled in based on the module's *spec*, before the *loader* executes and loads the module.

To create a module dynamically rather than using the import system, it's recommended to use `importlib.util.module_from_spec()`, which will set the various import-controlled attributes to appropriate values. It's also possible to use the `types.ModuleType` constructor to create modules directly, but this technique is more error-prone, as most attributes must be manually set on the module object after it has been created when using this approach.

조심

With the exception of *__name__*, it is **strongly** recommended that you rely on *__spec__* and its attributes instead of any of the other individual attributes listed in this subsection. Note that updating an attribute on `__spec__` will not update the corresponding attribute on the module itself:

```
>>> import typing
>>> typing.__name__, typing.__spec__.name
('typing', 'typing')
>>> typing.__spec__.name = 'spelling'
>>> typing.__name__, typing.__spec__.name
('typing', 'spelling')
>>> typing.__name__ = 'keyboard_smashing'
>>> typing.__name__, typing.__spec__.name
('keyboard_smashing', 'spelling')
```

`module.`**`__name__`**

The name used to uniquely identify the module in the import system. For a directly executed module, this will be set to `"__main__"`.

This attribute must be set to the fully qualified name of the module. It is expected to match the value of `module.__spec__.name`.

`module.`**`__spec__`**

A record of the module's import-system-related state.

Set to the `module spec` that was used when importing the module. See *Module specs* for more details.

Added in version 3.4.

`module.`**`__package__`**

The *package* a module belongs to.

If the module is top-level (that is, not a part of any specific package) then the attribute should be set to `''` (the empty string). Otherwise, it should be set to the name of the module's package (which can be equal to *module.__name__* if the module itself is a package). See **PEP 366** for further details.

This attribute is used instead of *__name__* to calculate explicit relative imports for main modules. It defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a `str`.

It is **strongly** recommended that you use `module.__spec__.parent` instead of `module.__package__`. *__package__* is now only used as a fallback if `__spec__.parent` is not set, and this fallback path is deprecated.

버전 3.4에서 변경: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

버전 3.6에서 변경: The value of `__package__` is expected to be the same as `__spec__.parent`. *__package__* is now only used as a fallback during import resolution if `__spec__.parent` is not defined.

버전 3.10에서 변경: `ImportWarning` is raised if an import resolution falls back to `__package__` instead of `__spec__.parent`.

버전 3.12에서 변경: Raise `DeprecationWarning` instead of `ImportWarning` when falling back to `__package__` during import resolution.

Deprecated since version 3.13, will be removed in version 3.15: `__package__` will cease to be set or taken into consideration by the import system or standard library.

`module.__loader__`

The *loader* object that the import machinery used to load the module.

This attribute is mostly useful for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

`__loader__` defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a *loader* object.

It is **strongly** recommended that you use `module.__spec__.loader` instead of `module.__loader__`.

버전 3.4에서 변경: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

Deprecated since version 3.12, will be removed in version 3.16: Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.

`module.__path__`

A (possibly empty) *sequence* of strings enumerating the locations where the package' s submodules will be found. Non-package modules should not have a `__path__` attribute. See *__path__ attributes on modules* for more details.

It is **strongly** recommended that you use `module.__spec__.submodule_search_locations` instead of `module.__path__`.

`module.__file__`

`module.__cached__`

`__file__` and `__cached__` are both optional attributes that may or may not be set. Both attributes should be a `str` when they are available.

`__file__` indicates the pathname of the file from which the module was loaded (if loaded from a file), or the pathname of the shared library file for extension modules loaded dynamically from a shared library. It might be missing for certain types of modules, such as C modules that are statically linked into the interpreter, and the *import system* may opt to leave it unset if it has no semantic meaning (for example, a module loaded from a database).

If `__file__` is set then the `__cached__` attribute might also be set, which is the path to any compiled version of the code (for example, a byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file *would* exist (see **PEP 3147**).

Note that `__cached__` may be set even if `__file__` is not set. However, that scenario is quite atypical. Ultimately, the *loader* is what makes use of the module spec provided by the *finder* (from which `__file__` and `__cached__` are derived). So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

It is **strongly** recommended that you use `module.__spec__.cached` instead of `module.__cached__`.

Deprecated since version 3.13, will be removed in version 3.15: Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, `__cached__` will cease to be set or taken into consideration by the import system or standard library.

Other writable attributes on module objects

As well as the import-related attributes listed above, module objects also have the following writable attributes:

`module.__doc__`

The module' s documentation string, or `None` if unavailable. See also: `__doc__ attributes`.

`module.__annotations__`

A dictionary containing *variable annotations* collected during module body execution. For best practices on working with `__annotations__`, see `annotationlib`.

버전 3.14에서 변경: Annotations are now *lazily evaluated*. See **PEP 649**.

`module.__annotate__`

The *annotate function* for this module, or `None` if the module has no annotations. See also: *`__annotate__`* attributes.

Added in version 3.14.

Module dictionaries

Module objects also have the following special read-only attribute:

`module.__dict__`

The module' s namespace as a dictionary object. Uniquely among the attributes listed here, `__dict__` cannot be accessed as a global variable from within a module; it can only be accessed as an attribute on module objects.

CPython 이 모듈 딕셔너리를 비우는 방법 때문에, 딕셔너리에 대한 참조가 남아있더라도, 모듈이 스코프를 벗어나면 모듈 딕셔너리는 비워집니다. 이것을 피하려면, 딕셔너리를 복사하거나 딕셔너리를 직접 이용하는 동안은 모듈을 잡아두어야 합니다.

3.2.10 사용자 정의 클래스(Custom classes)

Custom class types are typically created by class definitions (see section 클래스 정의). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., `C.x` is translated to `C.__dict__["x"]` (although there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. This search of the base classes uses the C3 method resolution order which behaves correctly even in the presence of 'diamond' inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by Python can be found at python_2.3_mro.

When a class attribute reference (for class `C`, say) would yield a class method object, it is transformed into an instance method object whose *`__self__`* attribute is `C`. When it would yield a `staticmethod` object, it is transformed into the object wrapped by the static method object. See section 디스크립터 구현하기 for another way in which attributes retrieved from a class may differ from those actually contained in its *`__dict__`*.

클래스 어트리뷰트 대입은 클래스의 딕셔너리를 갱신할 뿐, 어떤 경우도 부모 클래스의 딕셔너리를 건드리지는 않습니다.

클래스 객체는 클래스 인스턴스를 돌려주도록(아래를 보십시오) 호출될 수 있습니다(위를 보십시오).

Special attributes

어트리뷰트	의미
type.__name__	The class' s name. See also: __name__ attributes.
type.__qualname__	The class' s *qualified name*. See also: __qualname__ attributes.
type.__module__	The name of the module in which the class was defined.
type.__dict__	A mapping proxy providing a read-only view of the class' s namespace. See also: *__dict__ attributes*.
type.__bases__	A tuple containing the class' s bases. In most cases, for a class defined as class X(A, B, C), X.__bases__ will be exactly equal to (A, B, C).
type.__doc__	The class' s documentation string, or None if undefined. Not inherited by subclasses.
type.__annotations__	A dictionary containing *variable annotations* collected during class body execution. See also: *__annotations__ attributes*. For best practices on working with *__annotations__*, please see annotationlib. **조심** Accessing the __annotations__ attribute of a class object directly may yield incorrect results in the presence of metaclasses. In addition, the attribute may not exist for some classes. Use annotationlib.get_annotations() to retrieve class annotations safely. 버전 3.14에서 변경: Annotations are now *lazily evaluated*. See **PEP 649**.
type.__annotate__()	The *annotate function* for this class, or None if the class has no annotations. See also: *__annotate__ attributes*. **조심** Accessing the __annotate__ attribute of a class object directly may yield incorrect results in the presence of metaclasses. Use annotationlib.get_annotate_function() to retrieve the annotate function safely. Added in version 3.14.
type.__type_params__	A tuple containing the *type parameters* of a *generic class*. Added in version 3.12.
type.__static_attributes__	A tuple containing names of attributes of this class which are assigned through self.X from any function in its body. Added in version 3.13.
type.__firstlineno__	The line number of the first line of the class definition, including decorators. Setting the __module__ attribute removes the __firstlineno__ item from the type' s dictionary. Added in version 3.13.
type.__mro__	The tuple of classes that are considered when looking for base classes during method resolution.

Special methods

In addition to the special attributes described above, all Python classes also have the following two methods available:

type.**mro**()

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in *__mro__*.

type.**__subclasses__**()

Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. The list is in definition order. Example:

```
>>> class A: pass
>>> class B(A): pass
>>> A.__subclasses__()
[<class 'B'>]
```

3.2.11 클래스 인스턴스(Class instances)

A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object, it is transformed into an instance method object whose *__self__* attribute is the instance. Static method and class method objects are also transformed; see above under "Classes". See section 디스크립터 구현하기 for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's *__dict__*. If no class attribute is found, and the object's class has a *__getattr__()* method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a *__setattr__()* or *__delattr__()* method, this is called instead of updating the instance dictionary directly.

어떤 특별한 이름들의 메서드들을 가지면, 클래스 인스턴스는 숫자, 시퀀스, 매핑인 척할 수 있습니다. 특수 메서드 이름들 섹션을 보십시오.

Special attributes

object.**__class__**

The class to which a class instance belongs.

object.**__dict__**

A dictionary or other mapping object used to store an object's (writable) attributes. Not all instances have a `__dict__` attribute; see the section on *__slots__* for more details.

3.2.12 I/O 객체 (파일 객체라고도 알려져 있습니다)

파일 객체 는 열린 파일을 나타냅니다. 파일 객체를 만드는 여러 가지 단축법이 있습니다: `open()` 내장 함수, `os.popen()`, `os.fdopen()` 과 소켓 객체의 `makefile()` 메서드 (그리고, 아마도 확장 모듈들이 제공하는 다른 함수들이나 메서드들).

`sys.stdin`, `sys.stdout`, `sys.stderr` 는 인터프리터의 표준 입력, 출력, 에러 스트림으로 초기화된 파일 객체들입니다; 모두 텍스트 모드로 열려서 `io.TextIOBase` 추상 클래스에 의해 정의된 인터페이스를 따릅니다.

3.2.13 내부 형(Internal types)

인터프리터가 내부적으로 사용하는 몇몇 형들은 사용자에게 노출됩니다. 인터프리터의 미래 버전에서 이들의 정의는 변경될 수 있지만, 완전함을 위해 여기서 언급합니다.

코드 객체(Code objects)

코드 객체는 바이트로 컴파일된(*byte-compiled*) 실행 가능한 파이썬 코드를 나타내는데, 그냥 바이트 코드라고도 부릅니다. 코드 객체와 함수 객체 간에는 차이가 있습니다; 함수 객체는 함수의 전역 공간(globals)(함수가 정의된 모듈)을 명시적으로 참조하고 있지만, 코드 객체는 어떤 문맥(context)도 갖고 있지 않습니다; 또한 기본 인자값들이 함수 객체에 저장되어 있지만 코드 객체에는 들어있지 않습니다 (실행 시간에 계산되는 값들을 나타내기 때문입니다). 함수 객체와는 달리, 코드 객체는 불변이고 가변 객체들에 대한 어떤 참조도 (직접 혹은 간접적으로도) 갖고 있지 않습니다.

Special read-only attributes

codeobject.**co_name**	The function name
codeobject.**co_qualname**	The fully qualified function name Added in version 3.11.
codeobject.**co_argcount**	The total number of positional *parameters* (including positional-only parameters and parameters with default values) that the function has
codeobject.**co_posonlyargcount**	The number of positional-only *parameters* (including arguments with default values) that the function has
codeobject.**co_kwonlyargcount**	The number of keyword-only *parameters* (including arguments with default values) that the function has
codeobject.**co_nlocals**	The number of *local variables* used by the function (including parameters)
codeobject.**co_varnames**	A `tuple` containing the names of the local variables in the function (starting with the parameter names)
codeobject.**co_cellvars**	A `tuple` containing the names of *local variables* that are referenced from at least one *nested scope* inside the function
codeobject.**co_freevars**	A `tuple` containing the names of *free* (*closure*) *variables* that a *nested scope* references in an outer scope. See also *`function.__closure__`*. Note: references to global and builtin names are *not* included.
codeobject.**co_code**	A string representing the sequence of *bytecode* instructions in the function
codeobject.**co_consts**	A `tuple` containing the literals used by the *bytecode* in the function
codeobject.**co_names**	A `tuple` containing the names used by the *bytecode* in the function
codeobject.**co_filename**	The name of the file from which the code was compiled
codeobject.**co_firstlineno**	The line number of the first line of the function
codeobject.**co_lnotab**	A string encoding the mapping from *bytecode* offsets to line numbers. For details, see the source code of the interpreter. 버전 3.12부터 폐지됨: This attribute of code objects is deprecated, and may be removed in Python 3.15.
codeobject.**co_stacksize**	The required stack size of the code object
codeobject.**co_flags**	An `integer` encoding a number of flags for the interpreter.

The following flag bits are defined for *co_flags*: bit `0x04` is set if the function uses the `*arguments` syntax to accept an arbitrary number of positional arguments; bit `0x08` is set if the function uses the `**keywords` syntax to accept arbitrary keyword arguments; bit `0x20` is set if the function is a generator. See inspect-module-co-flags for details on the semantics of each flags that might be present.

Future feature declarations (for example, `from __future__ import division`) also use bits in *co_flags* to indicate whether a code object was compiled with a particular feature enabled. See `compiler_flag`.

Other bits in *co_flags* are reserved for internal use.

If a code object represents a function and has a docstring, the `CO_HAS_DOCSTRING` bit is set in *co_flags* and the first item in *co_consts* is the docstring of the function.

Methods on code objects

`codeobject.`**`co_positions`**`()`

Returns an iterable over the source code positions of each *bytecode* instruction in the code object.

The iterator returns `tuples` containing the `(start_line, end_line, start_column, end_column)`. The *i-th* tuple corresponds to the position of the source code that compiled to the *i-th* code unit. Column information is 0-indexed utf-8 byte offsets on the given source line.

This positional information can be missing. A non-exhaustive lists of cases where this may happen:

- Running the interpreter with `-X no_debug_ranges`.
- Loading a pyc file compiled while using `-X no_debug_ranges`.
- Position tuples corresponding to artificial instructions.
- Line and column numbers that can' t be represented due to implementation specific limitations.

When this occurs, some or all of the tuple elements can be `None`.

Added in version 3.11.

> **참고**
>
> This feature requires storing column positions in code objects which may result in a small increase of disk usage of compiled Python files or interpreter memory usage. To avoid storing the extra information and/or deactivate printing the extra traceback information, the `-X no_debug_ranges` command line flag or the `PYTHONNODEBUGRANGES` environment variable can be used.

`codeobject.`**`co_lines`**`()`

Returns an iterator that yields information about successive ranges of *bytecode*s. Each item yielded is a `(start, end, lineno) tuple`:

- `start` (an `int`) represents the offset (inclusive) of the start of the *bytecode* range
- `end` (an `int`) represents the offset (exclusive) of the end of the *bytecode* range
- `lineno` is an `int` representing the line number of the *bytecode* range, or `None` if the bytecodes in the given range have no line number

The items yielded will have the following properties:

- The first range yielded will have a `start` of 0.
- The `(start, end)` ranges will be non-decreasing and consecutive. That is, for any pair of `tuples`, the `start` of the second will be equal to the `end` of the first.
- No range will be backwards: `end >= start` for all triples.
- The last `tuple` yielded will have `end` equal to the size of the *bytecode*.

Zero-width ranges, where `start == end`, are allowed. Zero-width ranges are used for lines that are present in the source code, but have been eliminated by the *bytecode* compiler.

Added in version 3.10.

더 보기

PEP 626 - Precise line numbers for debugging and other tools.
The PEP that introduced the `co_lines()` method.

`codeobject.`**`replace`**`(`*`**kwargs`*`)`

Return a copy of the code object with new values for the specified fields.

Code objects are also supported by the generic function `copy.replace()`.

Added in version 3.8.

프레임 객체(Frame objects)

Frame objects represent execution frames. They may occur in *traceback objects*, and are also passed to registered trace functions.

Special read-only attributes

`frame.`**`f_back`**	Points to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame
`frame.`**`f_code`**	The *code object* being executed in this frame. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"f_code"`.
`frame.`**`f_locals`**	The mapping used by the frame to look up *local variables*. If the frame refers to an *optimized scope*, this may return a write-through proxy object. 버전 3.13에서 변경: Return a proxy for optimized scopes.
`frame.`**`f_globals`**	The dictionary used by the frame to look up *global variables*
`frame.`**`f_builtins`**	The dictionary used by the frame to look up *built-in (intrinsic) names*
`frame.`**`f_lasti`**	The “precise instruction” of the frame object (this is an index into the *bytecode* string of the *code object*)

Special writable attributes

`frame.`**`f_trace`**	If not `None`, this is a function called for various events during code execution (this is used by debuggers). Normally an event is triggered for each new source line (see *`f_trace_lines`*).
`frame.`**`f_trace_lines`**	Set this attribute to `False` to disable triggering a tracing event for each source line.
`frame.`**`f_trace_opcodes`**	Set this attribute to `True` to allow per-opcode events to be requested. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.
`frame.`**`f_lineno`**	The current line number of the frame – writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to this attribute.

Frame object methods

프레임 객체는 한가지 메서드를 지원합니다:

`frame.`**`clear`**`()`

This method clears all references to *local variables* held by the frame. Also, if the frame belonged to a *generator*, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its *traceback* for later use).

`RuntimeError` is raised if the frame is currently executing or suspended.

Added in version 3.4.

버전 3.13에서 변경: Attempting to clear a suspended frame raises `RuntimeError` (as has always been the case for executing frames).

트레이스백 객체(Traceback objects)

Traceback objects represent the stack trace of an exception. A traceback object is implicitly created when an exception occurs, and may also be explicitly created by calling `types.TracebackType`.

버전 3.7에서 변경: Traceback objects can now be explicitly instantiated from Python code.

For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section *try* 문.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

For explicitly created tracebacks, it is up to the creator of the traceback to determine how the *`tb_next`* attributes should be linked to form a full stack trace.

Special read-only attributes:

traceback.**tb_frame**	Points to the execution *frame* of the current level. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"tb_frame"`.
traceback.**tb_lineno**	Gives the line number where the exception occurred
traceback.**tb_lasti**	Indicates the "precise instruction".

The line number and last instruction in the traceback may differ from the line number of its *frame object* if the exception occurred in a `try` statement with no matching except clause or with a `finally` clause.

traceback.**tb_next**

The special writable attribute `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level.

버전 3.7에서 변경: This attribute is now writable

슬라이스 객체(Slice objects)

Slice objects are used to represent slices for `__getitem__()` methods. They are also created by the built-in `slice()` function.

특수 읽기 전용 어트리뷰트들: `start` 는 하한(lower bound) 입니다; `stop` 은 상한(upper bound) 입니다; `step` 은 스텝 값입니다; 각 값은 생략될 경우 `None` 입니다. 이 어트리뷰트들은 임의의 형이 될 수 있습니다.

슬라이스 객체는 하나의 메서드를 지원합니다.

slice.**indices**(*self*, *length*)

이 메서드는 하나의 정수 인자 *length* 를 받아서 슬라이스 객체가 길이 *length* 인 시퀀스에 적용되었을 때 그 슬라이스에 대한 정보를 계산합니다. 세 개의 정수로 구성된 튜플을 돌려줍니다: 이것들은 각각 *start* 와 *stop* 인덱스와, *step* 또는 슬라이스의 스트라이드(stride) 길이입니다. 생략되었거나 범위를 벗어난 인덱스들은 일반적인 슬라이스와 같은 방법으로 다뤄집니다.

스태틱 메서드 객체(Static method objects)

Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are also callable. Static method objects are created by the built-in `staticmethod()` constructor.

클래스 메서드 객체(Class method objects)

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under "*instance methods*". Class method objects are created by the built-in `classmethod()` constructor.

3.3 특수 메서드 이름들

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python' s approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `type(x).__getitem__(x, i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

Setting a special method to `None` indicates that the corresponding operation is not available. For example, if a class sets *__iter__()* to `None`, the class is not iterable, so calling `iter()` on its instances will raise a `TypeError` (without falling back to *__getitem__()*).[2]

내장형을 흉내 내는 클래스를 구현할 때, 모방은 모형화하는 객체에 말이 되는 수준까지만 구현하는 것이 중요합니다. 예를 들어, 어떤 시퀀스는 개별 항목들을 꺼내는 것만으로도 잘 동작할 수 있습니다. 하지만 슬라이스를 꺼내는 것은 말이 안 될 수 있습니다. (이런 한가지 예는 W3C의 Document Object Model의 `NodeList` 인터페이스입니다.)

3.3.1 기본적인 커스터마이제이션

`object.`**`__new__`**`(`*cls*`[, ...])`

클래스 *cls* 의 새 인스턴스를 만들기 위해 호출됩니다. *__new__()* 는 스태틱 메서드입니다 (그렇게 선언하지 않아도 되는 특별한 경우입니다) 인데, 첫 번째 인자로 만들려고 하는 인스턴스의 클래스가 전달됩니다. 나머지 인자들은 객체 생성자 표현 (클래스 호출) 에 전달된 것들입니다. *__new__()* 의 반환 값은 새 객체 인스턴스이어야 합니다 (보통 *cls* 의 인스턴스).

Typical implementations create a new instance of the class by invoking the superclass' s *__new__()* method using `super().__new__(cls[, ...])` with appropriate arguments and then modifying the newly created instance as necessary before returning it.

If *__new__()* is invoked during object construction and it returns an instance of *cls*, then the new instance' s *__init__()* method will be invoked like `__init__(self[, ...])`, where *self* is the new instance and the remaining arguments are the same as were passed to the object constructor.

만약 *__new__()* 가 *cls* 의 인스턴스를 돌려주지 않으면, 새 인스턴스의 *__init__()* 는 호출되지 않습니다.

__new__() 는 주로 불변형 (int, str, tuple과 같은) 의 서브 클래스가 인스턴스 생성을 커스터마이즈할 수 있도록 하는 데 사용됩니다. 또한, 사용자 정의 메타 클래스에서 클래스 생성을 커스터마이즈하기 위해 자주 사용됩니다.

`object.`**`__init__`**`(`*self*`[, ...])`

(*__new__()* 에 의해) 인스턴스가 만들어진 후에, 하지만 호출자에게 돌려주기 전에 호출됩니다. 인자들은 클래스 생성자 표현으로 전달된 것들입니다. 만약 베이스 클래스가 *__init__()* 메서드를 갖고 있다면, 서브 클래스의 *__init__()* 메서드는, 있다면, 인스턴스에서 베이스 클래스가 차지하는 부분이 올바르게 초기화됨을 확실히 하기 위해 명시적으로 호출해주어야 합니다; 예를 들어: `super().__init__([args...])`.

객체를 만드는데 *__new__()* 와 *__init__()* 가 협력하고 있으므로 (*__new__()* 는 만들고, *__init__()* 는 그것을 커스터마이즈합니다), *__init__()* 가 `None` 이외의 값을 돌려주면 실행시간에 `TypeError` 를 일으킵니다.

`object.`**`__del__`**`(`*self*`)`

인스턴스가 파괴되기 직전에 호출됩니다. 파이널라이저 또는 (부적절하게) 파괴자라고 불립니다. 만약 베이스 클래스가 *__del__()* 메서드를 갖고 있다면, 자식 클래스의 *__del__()* 메서드는, 정의되어 있다면, 인스턴스에서 베이스 클래스가 차지하는 부분을 적절하게 삭제하기 위해, 명시적으로 베이스 클래스의 메서드를 호출해야 합니다.

(권장하지는 않지만!) *__del__()* 메서드는 인스턴스에 대한 새로운 참조를 만듦으로써 인스턴스의 파괴를 지연시킬 수 있습니다. 이것을 객체 부활 이라고 부릅니다. 부활한 객체가 파괴될 때 *__del__()* 이 두 번째로 호출될지는 구현에 따라 다릅니다; 현재 *CPython* 구현은 오직 한 번만 호출합니다.

It is not guaranteed that *__del__()* methods are called for objects that still exist when the interpreter exits. `weakref.finalize` provides a straightforward way to register a cleanup function to be called when an object is garbage collected.

[2] The *__hash__()*, *__iter__()*, *__reversed__()*, *__contains__()*, *__class_getitem__()* and `__fspath__()` methods have special handling for this. Others will still raise a `TypeError`, but may do so by relying on the behavior that `None` is not callable.

참고

`del x` 는 직접 `x.__del__()` 를 호출하지 않습니다 — 앞에 있는 것은 `x` 의 참조 횟수(reference count)를 하나 감소시키고, 뒤에 있는 것은 `x` 의 참조 횟수가 0 이 될 때 호출됩니다.

CPython 구현 상세: It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by the *cyclic garbage collector*. A common cause of reference cycles is when an exception has been caught in a local variable. The frame' s locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

더 보기

`gc` 모듈에 대한 문서.

경고

__del__() 이 호출되는 불안정한 상황 때문에, 이것이 실행 중에 발생시키는 예외는 무시되고, 대신에 `sys.stderr` 로 경고가 출력됩니다. 특히:

- *__del__()* 은 (임의의 스레드에서) 임의의 코드가 실행되는 동안 호출될 수 있습니다. *__del__()* 이 록을 얻어야 하거나 다른 블로킹 자원을 호출하면, *__del__()* 을 실행하기 위해 중단된 코드가 자원을 이미 차지했을 수 있으므로 교착 상태에 빠질 수 있습니다.
- *__del__()* 은 인터프리터를 종료할 때 실행될 수 있습니다. 결과적으로, 액세스해야 하는 전역 변수(다른 모듈 포함)가 이미 삭제되었거나 `None` 으로 설정되었을 수 있습니다. 파이썬은 이름이 하나의 밑줄로 시작하는 전역 객체가 다른 전역 객체들보다 먼저 삭제됨을 보장합니다; 이것은, 만약 그 전역 객체들에 대한 다른 참조가 존재하지 않는다면, *__del__()* 메서드가 호출되는 시점에, 임포트된 모듈들이 남아있도록 확실히 하는 데 도움이 될 수 있습니다.

`object.`**`__repr__`**`(`*self*`)`

`repr()` 내장 함수에 의해 호출되어 객체의 "형식적인(official)" 문자열 표현을 계산합니다. 만약 가능하다면, 이것은 같은 (적절한 환경이 주어질 때) 값을 갖는 객체를 새로 만들 수 있는 올바른 파이썬 표현식처럼 보여야 합니다. 가능하지 않다면, `<...쓸 모 있는 설명...>` 형태의 문자열을 돌려줘야 합니다. 반환 값은 반드시 문자열이어야 합니다. 만약 클래스가 *__str__()* 없이 *__repr__()* 만 정의한다면, *__repr__()* 은 그 클래스 인스턴스의 "비형식적인(informal)" 문자열 표현이 요구될 때 사용될 수 있습니다.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous. A default implementation is provided by the `object` class itself.

`object.`**`__str__`**`(`*self*`)`

Called by `str(object)`, the default *__format__()* implementation, and the built-in function `print()`, to compute the "informal" or nicely printable string representation of an object. The return value must be a str object.

이 메서드는 *__str__()* 이 올바른 파이썬 표현식을 돌려줄 것이라고 기대되지 않는다는 점에서 *object.__repr__()* 과 다릅니다: 더 편리하고 간결한 표현이 사용될 수 있습니다.

내장형 `object` 에 정의된 기본 구현은 *object.__repr__()* 을 호출합니다.

`object.`**`__bytes__`**`(`*self*`)`

Called by bytes to compute a byte-string representation of an object. This should return a `bytes` object. The `object` class itself does not provide this method.

object.**__format__**(*self*, *format_spec*)

format() 내장 함수, 확대하면, 포맷 문자열 리터럴(*formatted string literals*) 의 계산과 str.format() 메서드에 의해 호출되어, 객체의 "포맷된" 문자열 표현을 만들어냅니다. *format_spec* 인자는 요구되는 포맷 옵션들을 포함하는 문자열입니다. *format_spec* 인자의 해석은 *__format__()* 을 구현하는 형에 달려있으나, 대부분 클래스는 포매팅을 내향형들의 하나로 위임하거나, 비슷한 포맷 옵션 문법을 사용합니다.

표준 포매팅 문법에 대해서는 formatspec 를 참고하면 됩니다.

반환 값은 반드시 문자열이어야 합니다.

The default implementation by the object class should be given an empty *format_spec* string. It delegates to *__str__()*.

버전 3.4에서 변경: object 의 __format__ 메서드 자신은, 빈 문자열이 아닌 인자가 전달되면 TypeError 를 발생시킵니다.

버전 3.7에서 변경: 이제 object.__format__(x, '') 는 format(str(x), '') 가 아니라 str(x) 와 동등합니다.

object.**__lt__**(*self*, *other*)

object.**__le__**(*self*, *other*)

object.**__eq__**(*self*, *other*)

object.**__ne__**(*self*, *other*)

object.**__gt__**(*self*, *other*)

object.**__ge__**(*self*, *other*)

이것들은 소위 "풍부한 비교(rich comparison)" 메서드입니다. 연산자 기호와 메서드 이름 간의 관계는 다음과 같습니다: x<y 는 x.__lt__(y) 를 호출합니다, x<=y 는 x.__le__(y) 를 호출합니다, x==y 는 x.__eq__(y) 를 호출합니다, x!=y 는 x.__ne__(y) 를 호출합니다, x>y 는 x.__gt__(y) 를 호출합니다, x>=y 는 x.__ge__(y) 를 호출합니다.

A rich comparison method may return the singleton NotImplemented if it does not implement the operation for a given pair of arguments. By convention, False and True are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an if statement), Python will call bool() on the value to determine if the result is true or false.

By default, object implements *__eq__()* by using is, returning NotImplemented in the case of a false comparison: True if x is y else NotImplemented. For *__ne__()*, by default it delegates to *__eq__()* and inverts the result unless it is NotImplemented. There are no other implied relationships among the comparison operators or default implementations; for example, the truth of (x<y or x==y) does not imply x<=y. To automatically generate ordering operations from a single root operation, see functools.total_ordering().

By default, the object class provides implementations consistent with 값 비교: equality compares according to object identity, and order comparisons raise TypeError. Each default method may generate these results directly, but may also return NotImplemented.

사용자 정의 비교 연산자를 지원하고 딕셔너리 키로 사용될 수 있는 해시 가능 객체를 만드는 것에 관한 몇 가지 중요한 내용이 *__hash__()* 에 관한 문단에 나옵니다.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, *__lt__()* and *__gt__()* are each other' s reflection, *__le__()* and *__ge__()* are each other' s reflection, and *__eq__()* and *__ne__()* are their own reflection. If the operands are of different types, and the right operand' s type is a direct or indirect subclass of the left operand' s type, the reflected method of the right operand has priority, otherwise the left operand' s method has priority. Virtual subclassing is not considered.

When no appropriate method returns any value other than NotImplemented, the == and != operators will fall back to is and is not, respectively.

object.**__hash__**(*self*)

Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. The `__hash__()` method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

참고

`hash()` 는 객체가 정의한 *`__hash__()`* 메서드가 돌려주는 값을 `Py_ssize_t` 의 크기로 자릅니다 (truncate). 이것은 보통 64-bit 빌드에서는 8바이트고, 32-bit 빌드에서는 4바이트입니다. 만약 객체의 *`__hash__()`* 가 서로 다른 비트 크기를 갖는 빌드들 사이에서 함께 사용되어야 한다면, 모든 지원할 빌드들에서의 폭을 검사해야 합니다. 이렇게 하는 쉬운 방법은 `python -c "import sys; print(sys.hash_info.width)"` 입니다.

If a class does not define an *`__eq__()`* method it should not define a *`__hash__()`* operation either; if it defines *`__eq__()`* but not *`__hash__()`*, its instances will not be usable as items in hashable collections. If a class defines mutable objects and implements an *`__eq__()`* method, it should not implement *`__hash__()`*, since the implementation of *hashable* collections requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

User-defined classes have *`__eq__()`* and *`__hash__()`* methods by default (inherited from the `object` class); with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns an appropriate value such that `x == y` implies both that `x is y` and `hash(x) == hash(y)`.

`__eq__()` 를 재정의하고 *`__hash__()`* 를 정의하지 않는 클래스는 *`__hash__()`* 가 `None` 으로 설정됩니다. 클래스의 *`__hash__()`* 메서드가 `None` 이면, 클래스의 인스턴스는 프로그램이 해시값을 얻으려 시도할 때 `TypeError` 를 일으키고, `isinstance(obj, collections.abc.Hashable)` 로 검사할 때 해시 가능하지 않다고 올바로 감지됩니다.

만약 *`__eq__()`* 를 재정의하는 클래스가 부모 클래스로부터 *`__hash__()`* 의 구현을 물려받고 싶으면 인터프리터에게 명시적으로 이렇게 지정해주어야 합니다: `__hash__ = <ParentClass>.__hash__`.

만약 *`__eq__()`* 를 재정의하지 않는 클래스가 해시 지원을 멈추고 싶으면, 클래스 정의에 `__hash__ = None` 을 포함해야 합니다. 자신의 *`__hash__()`* 을 정의한 후에 직접 `TypeError` 를 일으키는 경우는 `isinstance(obj, collections.abc.Hashable)` 호출이 해시 가능하다고 잘못 인식합니다.

참고

기본적으로, str과 bytes 객체들의 *`__hash__()`* 값은 예측할 수 없는 난수값으로 "솔트되어(salted)" 있습니다. 개별 파이썬 프로세스 내에서는 변하지 않는 값으로 유지되지만, 파이썬을 반복적으로 실행할 때는 예측할 수 없게 됩니다.

This is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict insertion, $O(n^2)$ complexity. See http://ocert.org/advisories/ocert-2011-003.html for details.

해시값의 변경은 집합의 이터레이션 순서에 영향을 줍니다, 파이썬은 이 순서에 대해 어떤 보장도 하지 않습니다 (그리고 보통 32-bit 와 64-bit 빌드 사이에서도 다릅니다).

`PYTHONHASHSEED` 를 참고하십시오.

버전 3.3에서 변경: 해시 난수 화는 기본적으로 활성화됩니다.

object.**__bool__**(*self*)

Called to implement truth value testing and the built-in operation `bool()`; should return `False` or `True`. When this method is not defined, *__len__()* is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither `__len__()` nor `__bool__()` (which is true of the `object` class itself), all its instances are considered true.

3.3.2 어트리뷰트 액세스 커스터마이제이션

클래스 인스턴스의 어트리뷰트 참조(읽기, 대입하기, `x.name` 을 삭제하기)의 의미를 변경하기 위해 다음과 같은 메서드들이 정의될 수 있습니다.

object.**__getattr__**(*self*, *name*)

Called when the default attribute access fails with an `AttributeError` (either *__getattribute__()* raises an `AttributeError` because *name* is not an instance attribute or an attribute in the class tree for `self`; or *__get__()* of a *name* property raises `AttributeError`). This method should either return the (computed) attribute value or raise an `AttributeError` exception. The `object` class itself does not provide this method.

Note that if the attribute is found through the normal mechanism, *__getattr__()* is not called. (This is an intentional asymmetry between *__getattr__()* and *__setattr__()*.) This is done both for efficiency reasons and because otherwise *__getattr__()* would have no way to access other attributes of the instance. Note that at least for instance variables, you can take total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the *__getattribute__()* method below for a way to actually get total control over attribute access.

object.**__getattribute__**(*self*, *name*)

클래스 인스턴스의 어트리뷰트 액세스를 구현하기 위해 조건 없이 호출됩니다. 만약 클래스가 *__getattr__()* 도 함께 구현하면, *__getattribute__()* 가 명시적으로 호출하거나 `AttributeError` 를 일으키지 않는 이상 *__getattr__* 는 호출되지 않습니다. 이 메서드는 어트리뷰트의 (계산된) 값을 돌려주거나 `AttributeError` 예외를 일으켜야 합니다. 이 메서드에서 무한 재귀(infinite recursion)가 발생하는 것을 막기 위해, 구현은 언제나 필요한 어트리뷰트에 접근하기 위해 같은 이름의 베이스 클래스의 메서드를 호출해야 합니다. 예를 들어, `object.__getattribute__(self, name)`.

> **참고**
>
> This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or *built-in functions*. See 특수 메서드 조회.

특정 민감한 어트리뷰트 액세스의 경우, 인자 `obj` 와 `name` 으로 감사 이벤트 `object.__getattr__` 을 발생시킵니다.

object.**__setattr__**(*self*, *name*, *value*)

어트리뷰트 대입이 시도될 때 호출됩니다. 일반적인 메커니즘(즉 인스턴스 딕셔너리에 값을 저장하는 것) 대신에 이것이 호출됩니다. *name* 은 어트리뷰트 이름이고, *value* 는 그것에 대입하려는 값입니다.

__setattr__() 에서 인스턴스 어트리뷰트에 대입하려고 할 때는, 같은 이름의 베이스 클래스의 메서드를 호출해야 합니다. 예를 들어 `object.__setattr__(self, name, value)`

특정 민감한 어트리뷰트 대입의 경우, 인자 `obj`, `name`, `value` 로 감사 이벤트 `object.__setattr__` 을 발생시킵니다.

object.**__delattr__**(*self*, *name*)

__setattr__() 과 비슷하지만 어트리뷰트를 대입하는 대신에 삭제합니다. 이것은 `del obj.name` 이 객체에 의미가 있는 경우에만 구현되어야 합니다.

특정 민감한 어트리뷰트 삭제의 경우, 인자 `obj` 와 `name` 으로 감사 이벤트 `object.__delattr__` 을 발생시킵니다.

object.**__dir__**(*self*)

Called when `dir()` is called on the object. An iterable must be returned. `dir()` converts the returned iterable to a list and sorts it.

모듈 어트리뷰트 액세스 커스터마이제이션

특수한 이름 `__getattr__` 과 `__dir__` 는 모듈 어트리뷰트에 대한 접근을 사용자 정의하는 데 사용될 수도 있습니다. 모듈 수준의 `__getattr__` 함수는 하나의 인자로 어트리뷰트의 이름을 받아서 계산된 값을 돌려주거나 `AttributeError` 를 발생시켜야 합니다. 일반적인 조회(즉 *object.__getattribute__()*)를 통해 어트리뷰트가 모듈 객체에서 발견되지 않으면, `AttributeError` 를 일으키기 전에 모듈 `__dict__` 에서 `__getattr__` 을 검색합니다. 발견되면, 어트리뷰트 이름으로 그 함수를 호출하고 결과를 돌려줍니다.

The `__dir__` function should accept no arguments, and return an iterable of strings that represents the names accessible on module. If present, this function overrides the standard `dir()` search on a module.

모듈 동작(어트리뷰트 설정, 프로퍼티 등)을 보다 세밀하게 사용자 정의하려면, 모듈 객체의 `__class__` 어트리뷰트를 `types.ModuleType` 의 서브 클래스로 설정할 수 있습니다. 예를 들면:

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

참고

모듈 `__getattr__` 정의와 모듈 `__class__` 설정은 어트리뷰트 액세스 구문을 사용하는 조회에만 영향을 미칩니다 – 모듈 전역에 대한 직접적인 액세스(모듈 내의 코드에 의한 액세스이거나 모듈의 전역 딕셔너리에 대한 참조를 거치거나)는 영향받지 않습니다.

버전 3.5에서 변경: 이제 `__class__` 모듈 어트리뷰트가 쓰기 가능합니다.

Added in version 3.7: `__getattr__` 과 `__dir__` 모듈 어트리뷰트.

더 보기

PEP 562 - 모듈 __getattr__ 과 __dir__
모듈에 대한 `__getattr__` 과 `__dir__` 함수를 설명합니다.

디스크립터 구현하기

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in an *owner* class (the descriptor must be in either the owner' s class dictionary or in the class dictionary for one of its parents). In the examples below, "the attribute" refers to the attribute whose name is the key of the property in the owner class' *__dict__*. The `object` class itself does not implement any of these protocols.

object.**__get__**(*self*, *instance*, *owner=None*)

소유자 클래스(클래스 어트리뷰트 액세스) 나 그 클래스의 인스턴스(인스턴스 어트리뷰트 액세스)의 어트리뷰트를 취하려고 할 때 호출됩니다. 선택적 *owner* 인자는 소유자 클래스입니다. 반면에 *instance* 는 어트리뷰트 참조가 일어나고 있는 인스턴스이거나, 어트리뷰트가 *owner* 를 통해 액세스되는 경우 *None* 입니다.

이 메서드는 계산된 어트리뷰트 값을 돌려주거나 AttributeError 예외를 일으켜야 합니다.

PEP 252는 *__get__()*이 하나나 두 개의 인자를 갖는 콜러블이라고 지정합니다. 파이썬 자신의 내장 디스크립터는 이 명세를 지원합니다; 그러나, 일부 제삼자 도구에는 두 인수를 모두 요구하는 디스크립터가 있을 수 있습니다. 파이썬 자신의 *__getattribute__()* 구현은 필요한지와 관계없이 항상 두 인자를 모두 전달합니다.

object.**__set__**(*self*, *instance*, *value*)

소유자 클래스의 인스턴스 *instance* 의 어트리뷰트를 새 값 *value* 로 설정할 때 호출됩니다.

*__set__()*이나 *__delete__()*를 추가하면 디스크립터 유형이 "데이터 디스크립터(data descriptor)"로 변경됨에 유의하십시오. 자세한 내용은 디스크립터 호출하기를 참조하십시오.

object.**__delete__**(*self*, *instance*)

소유자 클래스의 인스턴스 *instance* 의 어트리뷰트를 삭제할 때 호출됩니다.

Instances of descriptors may also have the __objclass__ attribute present:

object.**__objclass__**

The attribute __objclass__ is interpreted by the inspect module as specifying the class where this object was defined (setting this appropriately can assist in runtime introspection of dynamic class attributes). For callables, it may indicate that an instance of the given type (or a subclass) is expected or required as the first positional argument (for example, CPython sets this attribute for unbound methods that are implemented in C).

디스크립터 호출하기

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol: *__get__()*, *__set__()*, and *__delete__()*. If any of those methods are defined for an object, it is said to be a descriptor.

어트리뷰트 액세스의 기본 동작은 객체의 딕셔너리에서 어트리뷰트를 읽고, 쓰고, 삭제하는 것입니다. 예를 들어 a.x 는 a.__dict__['x'] 에서 시작해서 type(a).__dict__['x'] 를 거쳐 type(a) 의 메타 클래스를 제외한 베이스 클래스들을 거쳐 가는 일련의 조회로 구성됩니다.

그러나, 만약 조회한 값이 디스크립터 메서드를 구현한 객체면, 파이썬은 기본 동작 대신에 디스크립터 메서드를 호출할 수 있습니다. 우선순위 목록의 어느 위치에서 이런 일이 일어나는지는 어떤 디스크립터 메서드가 정의되어 있고 어떤 식으로 호출되는지에 따라 다릅니다.

디스크립터 호출의 시작점은 결합(binding)입니다, a.x. 어떻게 인자들이 조합되는지는 a 에 따라 다릅니다:

직접 호출
가장 간단하면서도 가장 덜 사용되는 호출은 사용자의 코드가 디스크립터 메서드를 직접 호출할 때입니다: x.__get__(a)

인스턴스 결합
객체 인스턴스에 결합하면, a.x 는 이런 호출로 변환됩니다: type(a).__dict__['x'].__get__(a, type(a)).

클래스 결합
클래스에 결합하면, A.x 는 이런 호출로 변환됩니다: A.__dict__['x'].__get__(None, A).

Super 결합
A dotted lookup such as super(A, a).x searches a.__class__.__mro__ for a base class B following A and then returns B.__dict__['x'].__get__(a, A). If not a descriptor, x is returned unchanged.

For instance bindings, the precedence of descriptor invocation depends on which descriptor methods are defined. A descriptor can define any combination of *__get__()*, *__set__()* and *__delete__()*. If it does not define __get__(), then accessing the attribute will return the descriptor object itself unless there is a value in the object's instance dictionary. If the descriptor defines __set__() and/or __delete__(), it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both __get__() and __set__(), while non-data descriptors have just the __get__() method. Data descriptors with __get__() and __set__() (and/or __delete__()) defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including those decorated with `@staticmethod` and `@classmethod`) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

`property()` 함수는 데이터 디스크립터로 구현됩니다. 이 때문에, 인스턴스는 프로퍼티(property)의 동작을 변경할 수 없습니다.

__slots__

__slots__ allow us to explicitly declare data members (like properties) and deny the creation of *__dict__* and *__weakref__* (unless explicitly declared in *__slots__* or available in a parent.)

The space saved over using *__dict__* can be significant. Attribute lookup speed can be significantly improved as well.

`object.`**`__slots__`**

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. *__slots__* reserves space for the declared variables and prevents the automatic creation of *__dict__* and *__weakref__* for each instance.

Notes on using *__slots__*:

- When inheriting from a class without *__slots__*, the *__dict__* and *__weakref__* attribute of the instances will always be accessible.
- Without a *__dict__* variable, instances cannot be assigned new variables not listed in the *__slots__* definition. Attempts to assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add `'__dict__'` to the sequence of strings in the *__slots__* declaration.
- Without a *__weakref__* variable for each instance, classes defining *__slots__* do not support `weak references` to its instances. If weak reference support is needed, then add `'__weakref__'` to the sequence of strings in the *__slots__* declaration.
- *__slots__* are implemented at the class level by creating *descriptors* for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by *__slots__*; otherwise, the class attribute would overwrite the descriptor assignment.
- The action of a *__slots__* declaration is not limited to the class where it is defined. *__slots__* declared in parents are available in child classes. However, instances of a child subclass will get a *__dict__* and *__weakref__* unless the subclass also defines *__slots__* (which should only contain names of any *additional* slots).
- 클래스가 베이스 클래스의 *__slots__* 에 정의된 이름과 같은 이름의 변수를 *__slots__* 에 선언한다면, 베이스 클래스가 정의한 변수는 액세스할 수 없는 상태가 됩니다(베이스 클래스로부터 디스크립터를 직접 조회하는 경우는 예외다). 이것은 프로그램을 정의되지 않은 상태로 보내게 됩니다. 미래에는, 이를 방지하기 위한 검사가 추가될 것입니다.
- `TypeError` will be raised if nonempty *__slots__* are defined for a class derived from a `"variable-length" built-in type` such as `int`, `bytes`, and `tuple`.
- Any non-string *iterable* may be assigned to *__slots__*.
- If a `dictionary` is used to assign *__slots__*, the dictionary keys will be used as the slot names. The values of the dictionary can be used to provide per-attribute docstrings that will be recognised by `inspect.getdoc()` and displayed in the output of `help()`.
- *__class__* assignment works only if both classes have the same *__slots__*.
- Multiple inheritance with multiple slotted parent classes can be used, but only one parent is allowed to have attributes created by slots (the other bases must have empty slot layouts) - violations raise `TypeError`.
- If an *iterator* is used for *__slots__* then a *descriptor* is created for each of the iterator's values. However, the *__slots__* attribute will be an empty iterator.

3.3.3 클래스 생성 커스터마이제이션

Whenever a class inherits from another class, *__init_subclass__()* is called on the parent class. This way, it is possible to write classes which change the behavior of subclasses. This is closely related to class decorators, but where class decorators only affect the specific class they' re applied to, `__init_subclass__` solely applies to future subclasses of the class defining the method.

classmethod `object.__init_subclass__`(*cls*)

이 메서드는 포함하는 클래스의 서브 클래스가 만들어질 때마다 호출됩니다. *cls* 는 새 서브 클래스입니다. 만약 일반적인 인스턴스 메서드로 정의되면, 이 메서드는 묵시적으로 클래스 메서드로 변경됩니다.

Keyword arguments which are given to a new class are passed to the parent class' s `__init_subclass__`. For compatibility with other classes using `__init_subclass__`, one should take out the needed keyword arguments and pass the others over to the base class, as in:

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name

class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

기본 구현 `object.__init_subclass__` 는 아무 일도 하지 않지만, 인자가 포함되어 호출되면 예외를 발생시킵니다.

> **참고**
>
> 메타 클래스 힌트 `metaclass` 는 나머지 형 절차에 의해 소비되고, `__init_subclass__` 로 전달되지 않습니다. 실제 메타 클래스 (명시적인 힌트 대신에) 는 `type(cls)` 로 액세스할 수 있습니다.

Added in version 3.6.

When a class is created, `type.__new__()` scans the class variables and makes callbacks to those with a *__set_name__()* hook.

`object.__set_name__`(*self*, *owner*, *name*)

Automatically called at the time the owning class *owner* is created. The object has been assigned to *name* in that class:

```
class A:
    x = C()  # Automatically calls: x.__set_name__(A, 'x')
```

If the class variable is assigned after the class is created, *__set_name__()* will not be called automatically. If needed, *__set_name__()* can be called directly:

```
class A:
   pass

c = C()
A.x = c                  # The hook is not called
c.__set_name__(A, 'x')   # Manually invoke the hook
```

더 자세한 내용은 클래스 객체 만들기 을 참고하십시오.

Added in version 3.6.

메타 클래스

기본적으로, 클래스는 `type()` 을 사용해서 만들어집니다. 클래스의 바디는 새 이름 공간에서 실행되고, 클래스 이름은 `type(name, bases, namespace)` 의 결과에 지역적으로 연결됩니다.

클래스를 만드는 과정은 클래스 정의 줄에 `metaclass` 키워드 인자를 전달하거나, 그런 인자를 포함한 이미 존재하는 클래스를 계승함으로써 커스터마이즈될 수 있습니다. 다음 예에서, `MyClass` 와 `MySubclass` 는 모두 `Meta` 의 인스턴스입니다.

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

클래스 정의에서 지정된 다른 키워드 인자들은 아래에서 설명되는 모든 메타 클래스 연산들로 전달됩니다.

클래스 정의가 실행될 때, 다음과 같은 단계가 수행됩니다.:

- MRO 항목이 결정됩니다;
- 적절한 메타 클래스가 결정됩니다;
- 클래스 이름 공간이 준비됩니다;
- 클래스 바디가 실행됩니다;
- 클래스 객체가 만들어집니다.

MRO 항목 결정하기

`object.__mro_entries__`(*self*, *bases*)

If a base that appears in a class definition is not an instance of `type`, then an `__mro_entries__()` method is searched on the base. If an `__mro_entries__()` method is found, the base is substituted with the result of a call to `__mro_entries__()` when creating the class. The method is called with the original bases tuple passed to the *bases* parameter, and must return a tuple of classes that will be used instead of the base. The returned tuple may be empty: in these cases, the original base is ignored.

더 보기

types.resolve_bases()
Dynamically resolve bases that are not instances of `type`.

types.get_original_bases()
Retrieve a class' s "original bases" prior to modifications by *__mro_entries__()*.

PEP 560
Core support for typing module and generic types.

적절한 메타 클래스 선택하기

클래스 정의의 적절한 메타 클래스는 다음과 같이 결정됩니다:

- 베이스와 명시적인 메타 클래스를 주지 않는 경우 `type()` 이 사용됩니다;
- 명시적인 메타 클래스가 지정되고, 그것이 `type()` 의 인스턴스가 아니면, 그것을 메타 클래스로 사용합니다;
- `type()` 의 인스턴스가 명시적인 메타 클래스로 주어지거나, 베이스가 정의되었으면, 가장 많이 파생된 메타 클래스가 사용됩니다.

가장 많이 파생된 메타 클래스는 명시적으로 지정된 메타 클래스(있다면)와 지정된 모든 베이스 클래스들의 메타 클래스들(즉, `type(cls)`) 중에서 선택됩니다. 가장 많이 파생된 메타 클래스는 이들 모두 의 서브 타입(subtype) 입니다. 만약 어느 것도 이 조건을 만족하지 못한다면, 클래스 정의는 `TypeError` 를 발생시키며 실패합니다.

클래스 이름 공간 준비하기

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwds)` (where the additional keyword arguments, if any, come from the class definition). The `__prepare__` method should be implemented as a `classmethod`. The namespace returned by `__prepare__` is passed in to `__new__`, but when the final class object is created the namespace is copied into a new `dict`.

만약 메타 클래스에 `__prepare__` 어트리뷰트가 없다면, 클래스 이름 공간은 빈 순서 있는 매핑으로 초기화됩니다.

더 보기

PEP 3115 - 파이썬 3000 에서의 메타 클래스
`__prepare__` 이름 공간 훅을 도입했습니다

클래스 바디 실행하기

클래스 바디는 (대략) `exec(body, globals(), namespace)` 과같이 실행됩니다. 일반적인 `exec()` 호출과 주된 차이점은 클래스 정의가 함수 내부에서 이루어질 때 어휘 스코핑(lexical scoping) 이 클래스 바디(모든 메서드들을 포함해서)로 하여금 현재와 외부 스코프에 있는 이름들을 참조하도록 허락한다는 것입니다.

하지만, 클래스 정의가 함수 내부에서 이루어질 때조차도, 클래스 내부에서 정의된 메서드들은 클래스 스코프에서 정의된 이름들을 볼 수 없습니다. 클래스 변수는 인스턴스나 클래스 메서드의 첫 번째 매개변수를 통해 액세스하거나 다음 섹션에서 설명하는 묵시적으로 어휘 스코핑된 `__class__` 참조를 통해야 합니다.

클래스 객체 만들기

일단 클래스 이름 공간이 클래스 바디를 실행함으로써 채워지면, 클래스 객체가 `metaclass(name, bases, namespace, **kwds)` 을 통해 만들어집니다(여기에서 전달되는 추가적인 키워드 인자들은 `__prepare__` 에 전달된 것들과 같습니다).

이 클래스 객체는 `super()` 에 인자를 주지 않는 경우 참조되는 것입니다. `__class__` 는 클래스 바디의 메서드들 중 어느 하나라도 `__class__` 나 `super` 를 참조할 경우 컴파일러에 의해 만들어지는 묵시적인 클로저(closure) 참조입니다. 이것은 인자 없는 형태의 `super()` 가 어휘 스코핑 기반으로 현재 정의되고 있는 클래스를 올바르게 찾을 수 있도록 합니다. 반면에 현재의 호출에 사용된 클래스나 인스턴스는 메서드로 전달된 첫 번째 인자에 기초해서 식별됩니다.

CPython 3.6 이상에서, `__class__` 셀(cell) 은 클래스 이름 공간의 `__classcell__` 엔트리로 메타 클래스에 전달됩니다. 만약 존재한다면, 이것은 클래스가 올바르게 초기화되기 위해 `type.__new__` 호출까지 거슬러서 전파되어야 합니다. 이렇게 하지 못하면 파이썬 3.8 에서는 `RuntimeError` 로 이어질 것입니다.

When using the default metaclass `type`, or any metaclass that ultimately calls `type.__new__`, the following additional customization steps are invoked after creating the class object:

1) The `type.__new__` method collects all of the attributes in the class namespace that define a *`__set_name__()`* method;
2) Those `__set_name__` methods are called with the class being defined and the assigned name of that particular attribute;
3) The *`__init_subclass__()`* hook is called on the immediate parent of the new class in its method resolution order.

클래스 객체가 만들어진 후에, 클래스 정의에 포함된 클래스 데코레이터들에게 (있다면) 클래스를 전달하고, 그 결과를 클래스가 정의되는 지역 이름 공간에 연결합니다.

When a new class is created by `type.__new__`, the object provided as the namespace parameter is copied to a new ordered mapping and the original object is discarded. The new copy is wrapped in a read-only proxy, which becomes the *__dict__* attribute of the class object.

더 보기

PEP 3135 - 새 super
묵시적인 __class__ 클로저 참조를 설명합니다

메타 클래스의 용도

메타 클래스의 잠재적인 용도에는 한계가 없습니다. 탐색 된 몇 가지 아이디어들에는 enum, 로깅, 인터페이스 검사, 자동화된 위임(automatic delegation), 자동화된 프로퍼티(properety) 생성, 프락시(proxy), 프레임웍(framework), 자동화된 자원 로킹/동기화(automatic resource locking/synchronization) 등이 있습니다.

3.3.4 인스턴스 및 서브 클래스 검사 커스터마이제이션

다음 메서드들은 `isinstance()` 와 `issubclass()` 내장 함수들의 기본 동작을 재정의하는 데 사용됩니다.

특히, 메타 클래스 `abc.ABCMeta` 는 추상 베이스 클래스(Abstract Base Class, ABC)를 다른 ABC를 포함한 임의의 클래스나 형(내장형을 포함합니다)에 "가상 베이스 클래스(virtual base class)"로 추가할 수 있게 하려고 이 메서드들을 구현합니다.

`type.`**`__instancecheck__`**(*self*, *instance*)

instance 가 (직접적이거나 간접적으로) *class* 의 인스턴스로 취급될 수 있으면 참을 돌려줍니다. 만약 정의되면, `isinstance(instance, class)` 를 구현하기 위해 호출됩니다.

`type.`**`__subclasscheck__`**(*self*, *subclass*)

subclass 가 (직접적이거나 간접적으로) *class* 의 서브 클래스로 취급될 수 있으면 참을 돌려줍니다. 만약 정의되면, `issubclass(subclass, class)` 를 구현하기 위해 호출됩니다.

이 메서드들은 클래스의 형(메타 클래스)에서 조회된다는 것에 주의해야 합니다. 실제 클래스에서 클래스 메서드로 정의될 수 없습니다. 이것은 인스턴스에 대해 호출되는 특수 메서드들의 조회와 일관성 있습니다. 이 경우 인스턴스는 클래스 자체다.

더 보기

PEP 3119 - 추상 베이스 클래스의 도입
Includes the specification for customizing `isinstance()` and `issubclass()` behavior through *__instancecheck__()* and *__subclasscheck__()*, with motivation for this functionality in the context of adding Abstract Base Classes (see the `abc` module) to the language.

3.3.5 제네릭 형 흉내 내기

When using *type annotations*, it is often useful to *parameterize* a *generic type* using Python's square-brackets notation. For example, the annotation `list[int]` might be used to signify a `list` in which all the elements are of type `int`.

더 보기

PEP 484 - Type Hints
Introducing Python's framework for type annotations

Generic Alias Types
Documentation for objects representing parameterized generic classes

Generics, user-defined generics and `typing.Generic`
Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

A class can *generally* only be parameterized if it defines the special class method `__class_getitem__()`.

classmethod `object.__class_getitem__`(*cls*, *key*)

key 에 있는 형 인자에 의한 제네릭 클래스의 특수화를 나타내는 객체를 돌려줍니다.

When defined on a class, `__class_getitem__()` is automatically a class method. As such, there is no need for it to be decorated with `@classmethod` when it is defined.

The purpose of *__class_getitem__*

The purpose of *`__class_getitem__()`* is to allow runtime parameterization of standard-library generic classes in order to more easily apply *type hints* to these classes.

To implement custom generic classes that can be parameterized at runtime and understood by static type-checkers, users should either inherit from a standard library class that already implements *`__class_getitem__()`*, or inherit from `typing.Generic`, which has its own implementation of `__class_getitem__()`.

Custom implementations of *`__class_getitem__()`* on classes defined outside of the standard library may not be understood by third-party type-checkers such as mypy. Using `__class_getitem__()` on any class for purposes other than type hinting is discouraged.

__class_getitem__ versus *__getitem__*

Usually, the *subscription* of an object using square brackets will call the *`__getitem__()`* instance method defined on the object' s class. However, if the object being subscribed is itself a class, the class method *`__class_getitem__()`* may be called instead. `__class_getitem__()` should return a GenericAlias object if it is properly defined.

Presented with the *expression* `obj[x]`, the Python interpreter follows something like the following process to decide whether *`__getitem__()`* or *`__class_getitem__()`* should be called:

```
from inspect import isclass

def subscribe(obj, x):
    """Return the result of the expression 'obj[x]'"""

    class_of_obj = type(obj)

    # If the class of obj defines __getitem__,
    # call class_of_obj.__getitem__(obj, x)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

    # Else, if obj is a class and defines __class_getitem__,
    # call obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # Else, raise an exception
    else:
        raise TypeError(
            f"'{class_of_obj.__name__}' object is not subscriptable"
        )
```

In Python, all classes are themselves instances of other classes. The class of a class is known as that class' s *metaclass*, and most classes have the `type` class as their metaclass. `type` does not define *`__getitem__()`*, meaning that expres-

sions such as `list[int]`, `dict[str, float]` and `tuple[str, bytes]` all result in *__class_getitem__()* being called:

```
>>> # list has class "type" as its metaclass, like most classes:
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) == type(bytes)
True
>>> # "list[int]" calls "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ returns a GenericAlias object:
>>> type(list[int])
<class 'types.GenericAlias'>
```

However, if a class has a custom metaclass that defines *__getitem__()*, subscribing the class may result in different behaviour. An example of this can be found in the `enum` module:

```
>>> from enum import Enum
>>> class Menu(Enum):
...     """A breakfast menu"""
...     SPAM = 'spam'
...     BACON = 'bacon'
...
>>> # Enum classes have a custom metaclass:
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta defines __getitem__,
>>> # so __class_getitem__ is not called,
>>> # and the result is not a GenericAlias object:
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

더 보기

PEP 560 - Core Support for typing module and generic types
Introducing *__class_getitem__()*, and outlining when a *subscription* results in `__class_getitem__()` being called instead of *__getitem__()*

3.3.6 콜러블 객체 흉내 내기

`object.`**`__call__`**`(`*self*`[`, *args...*`]`)

Called when the instance is "called" as a function; if this method is defined, `x(arg1, arg2, ...)` roughly translates to `type(x).__call__(x, arg1, ...)`. The `object` class itself does not provide this method.

3.3.7 컨테이너형 흉내 내기

The following methods can be defined to implement container objects. None of them are provided by the `object` class itself. Containers usually are *sequences* (such as `lists` or `tuples`) or *mappings* (like *dictionaries*), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= k < N` where *N* is the length of the sequence, or `slice` objects, which define a range of items. It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python' s standard `dictionary` objects. The

`collections.abc` module provides a `MutableMapping` *abstract base class* to help create those methods from a base set of *`__getitem__()`*, *`__setitem__()`*, *`__delitem__()`*, and `keys()`. Mutable sequences should provide methods `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` and `sort()`, like Python standard `list` objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods *`__add__()`*, *`__radd__()`*, *`__iadd__()`*, *`__mul__()`*, *`__rmul__()`* and *`__imul__()`* described below; they should not define other numerical operators. It is recommended that both mappings and sequences implement the *`__contains__()`* method to allow efficient use of the `in` operator; for mappings, `in` should search the mapping' s keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the *`__iter__()`* method to allow efficient iteration through the container; for mappings, `__iter__()` should iterate through the object' s keys; for sequences, it should iterate through the values.

`object.`**`__len__`**`(`*self*`)`

Called to implement the built-in function `len()`. Should return the length of the object, an integer >= 0. Also, an object that doesn' t define a *`__bool__()`* method and whose `__len__()` method returns zero is considered to be false in a Boolean context.

CPython 구현 상세: In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than `sys.maxsize` some features (such as `len()`) may raise `OverflowError`. To prevent raising `OverflowError` by truth value testing, an object must define a *`__bool__()`* method.

`object.`**`__length_hint__`**`(`*self*`)`

Called to implement `operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer >= 0. The return value may also be `NotImplemented`, which is treated the same as if the `__length_hint__` method didn' t exist at all. This method is purely an optimization and is never required for correctness.

Added in version 3.4.

참고

슬라이싱은 전적으로 다음에 나오는 세 메서드들에의해 수행됩니다

```
a[1:2] = b
```

과 같은 호출은

```
a[slice(1, 2, None)] = b
```

로 번역되고, 다른 형태도 마찬가지입니다. 빠진 슬라이스 항목은 항상 `None` 으로 채워집니다.

`object.`**`__getitem__`**`(`*self*`, `*key*`)`

Called to implement evaluation of `self[key]`. For *sequence* types, the accepted keys should be integers. Optionally, they may support `slice` objects as well. Negative index support is also optional. If *key* is of an inappropriate type, `TypeError` may be raised; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For *mapping* types, if *key* is missing (not in the container), `KeyError` should be raised.

참고

for 루프는 시퀀스의 끝을 올바로 감지하기 위해, 잘못된 인덱스에 대해 `IndexError` 가 일어날 것으로 기대하고 있습니다.

참고

When *subscripting* a *class*, the special class method *`__class_getitem__()`* may be called instead of `__getitem__()`. See *`__class_getitem__` versus `__getitem__`* for more details.

object.**__setitem__**(*self*, *key*, *value*)

`self[key]` 로의 대입을 구현하기 위해 호출됩니다. *__getitem__()* 과 같은 주의가 필요합니다. 매핑의 경우에는, 객체가 키에 대해 값의 변경이나 새 키의 추가를 허락할 경우, 시퀀스의 경우는 항목이 교체될 수 있을 때만 구현되어야 합니다. 잘못된 *key* 값의 경우는 *__getitem__()* 에서와 같은 예외를 일으켜야 합니다.

object.**__delitem__**(*self*, *key*)

`self[key]` 의 삭제를 구현하기 위해 호출됩니다. *__getitem__()* 과 같은 주의가 필요합니다. 매핑의 경우에는, 객체가 키의 삭제를 허락할 경우, 시퀀스의 경우는 항목이 시퀀스로부터 제거될 수 있을 때만 구현되어야 합니다. 잘못된 *key* 값의 경우는 *__getitem__()* 에서와 같은 예외를 일으켜야 합니다.

object.**__missing__**(*self*, *key*)

`dict.`*__getitem__()* 이 dict 서브 클래스에서 키가 딕셔너리에 없으면 `self[key]` 를 구현하기 위해 호출합니다.

object.**__iter__**(*self*)

This method is called when an *iterator* is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

object.**__reversed__**(*self*)

`reversed()` 내장 함수가 역 이터레이션(reverse iteration)을 구현하기 위해 (있다면) 호출합니다. 컨테이너에 있는 객체들을 역 순으로 탐색하는 새 이터레이터 객체를 돌려줘야 합니다.

__reversed__() 메서드가 제공되지 않으면, `reversed()` 내장함수는 시퀀스 프로토콜(*__len__()* 과 *__getitem__()*)을 대안으로 사용합니다. 시퀀스 프로토콜을 지원하는 객체들은 `reversed()` 가 제공하는 것보다 더 효율적인 구현을 제공할 수 있을 때만 *__reversed__()* 를 제공해야 합니다.

멤버십 검사 연산자들(*in* 과 *not in*) 은 보통 컨테이너에 대한 이터레이션으로 구현됩니다. 하지만, 컨테이너 객체는 더 효율적인 구현을 다음과 같은 특수 메서드를 통해 제공할 수 있습니다. 이 경우 객체는 이터러블일 필요도 없습니다.

object.**__contains__**(*self*, *item*)

멤버십 검사 연산자를 구현하기 위해 호출됩니다. *item* 이 *self* 에 있으면 참을, 그렇지 않으면 거짓을 돌려줘야 합니다. 매핑 객체의 경우, 키-값 쌍이 아니라 매핑의 키가 고려되어야 합니다.

__contains__() 를 정의하지 않는 객체의 경우, 멤버십 검사는 먼저 *__iter__()* 를 통한 이터레이션을 시도한 후, *__getitem__()* 을 통한 낡은 시퀀스 이터레이션 프로토콜을 시도합니다. 언어 레퍼런스의 이 절을 참고하십시오.

3.3.8 숫자 형 흉내 내기

숫자 형을 흉내 내기 위해 다음과 같은 메서드들을 정의할 수 있습니다. 구현되는 특별한 종류의 숫자에 의해 지원되지 않는 연산들(예를 들어, 정수가 아닌 숫자들에 대한 비트 연산들)에 대응하는 메서드들을 정의되지 않은 채로 남겨두어야 합니다.

object.**__add__**(*self*, *other*)

object.**__sub__**(*self*, *other*)

object.**__mul__**(*self*, *other*)

object.**__matmul__**(*self*, *other*)

object.**__truediv__**(*self*, *other*)

object.**__floordiv__**(*self*, *other*)

object.**__mod__**(*self*, *other*)

object.**__divmod__**(*self*, *other*)

object.**__pow__**(*self*, *other*[, *modulo*])

object.**__lshift__**(*self*, *other*)

object.**__rshift__**(*self*, *other*)

object.**__and__**(*self*, *other*)

object.**__xor__**(*self*, *other*)

object.**__or__**(*self*, *other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where *x* is an instance of a class that has an *__add__()* method, `type(x).__add__(x, y)` is called. The *__divmod__()* method should be the equivalent to using *__floordiv__()* and *__mod__()*; it should not be related to *__truediv__()*. Note that *__pow__()* should be defined to accept an optional third argument if the three-argument version of the built-in `pow()` function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return `NotImplemented`.

object.**__radd__**(*self*, *other*)

object.**__rsub__**(*self*, *other*)

object.**__rmul__**(*self*, *other*)

object.**__rmatmul__**(*self*, *other*)

object.**__rtruediv__**(*self*, *other*)

object.**__rfloordiv__**(*self*, *other*)

object.**__rmod__**(*self*, *other*)

object.**__rdivmod__**(*self*, *other*)

object.**__rpow__**(*self*, *other*[, *modulo*])

object.**__rlshift__**(*self*, *other*)

object.**__rrshift__**(*self*, *other*)

object.**__rand__**(*self*, *other*)

object.**__rxor__**(*self*, *other*)

object.**__ror__**(*self*, *other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the operands are of different types, when the left operand does not support the corresponding operation[3], or the right operand' s class is derived from the left operand' s class.[4] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an *__rsub__()* method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns `NotImplemented` or `type(y)` is a subclass of `type(x)`.[5]

Note that *__rpow__()* should be defined to accept an optional third argument if the three-argument version of the built-in `pow()` function is to be supported.

버전 3.14.0a7 (unreleased)에서 변경: Three-argument `pow()` now try calling *__rpow__()* if necessary. Previously it was only called in two-argument `pow()` and the binary power operator.

> **참고**
>
> 만약 오른쪽 피연산자의 형이 왼쪽 피연산자의 형의 서브 클래스이고, 그 서브 클래스가 연산의 뒤집힌 메서드의 다른 구현을 제공하면, 이 메서드가 왼쪽 연산자의 뒤집히지 않은 메서드보다 먼저 호출됩니다. 이 동작은 서브 클래스가 조상들의 연산을 재정의할 수 있도록 합니다.

object.**__iadd__**(*self*, *other*)

object.**__isub__**(*self*, *other*)

object.**__imul__**(*self*, *other*)

object.**__imatmul__**(*self*, *other*)

[3] "Does not support" here means that the class has no such method, or the method returns `NotImplemented`. Do not set the method to `None` if you want to force fallback to the right operand' s reflected method—that will instead have the opposite effect of explicitly *blocking* such fallback.

[4] For operands of the same type, it is assumed that if the non-reflected method (such as *__add__()*) fails then the operation is not supported, which is why the reflected method is not called.

[5] If the right operand' s type is a subclass of the left operand' s type, the reflected method having precedence allows subclasses to override their ancestors' operations.

object.**__itruediv__**(*self*, *other*)

object.**__ifloordiv__**(*self*, *other*)

object.**__imod__**(*self*, *other*)

object.**__ipow__**(*self*, *other*[, *modulo*])

object.**__ilshift__**(*self*, *other*)

object.**__irshift__**(*self*, *other*)

object.**__iand__**(*self*, *other*)

object.**__ixor__**(*self*, *other*)

object.**__ior__**(*self*, *other*)

These methods are called to implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `@=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, or if that method returns `NotImplemented`, the augmented assignment falls back to the normal methods. For instance, if *x* is an instance of a class with an *__iadd__()* method, `x += y` is equivalent to `x = x.__iadd__(y)` . If *__iadd__()* does not exist, or if `x.__iadd__(y)` returns `NotImplemented`, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`. In certain situations, augmented assignment can result in unexpected errors (see faq-augmented-assignment-tuple-error), but this behavior is in fact part of the data model.

object.**__neg__**(*self*)

object.**__pos__**(*self*)

object.**__abs__**(*self*)

object.**__invert__**(*self*)

일 항 산술 연산(`-`, `+`, `abs()`, `~`)을 구현하기 위해 호출됩니다.

object.**__complex__**(*self*)

object.**__int__**(*self*)

object.**__float__**(*self*)

내장 함수 `complex()`, `int()`, `float()`를 구현하기 위해 호출됩니다. 적절한 형의 값을 돌려줘야 합니다.

object.**__index__**(*self*)

`operator.index()`를 구현하기 위해 호출되고, 파이썬이 숫자 객체를 정수 객체로 손실 없이 변환해야 할 때(슬라이싱이나 내장 `bin()`, `hex()`, `oct()` 함수들에서와같이)마다 호출됩니다. 이 메서드의 존재는 숫자 객체가 정수 형임을 가리킵니다. 반드시 정수를 돌려줘야 합니다.

__int__(), *__float__()* 및 *__complex__()*가 정의되어 있지 않으면, 해당 내장 함수 `int()`, `float()` 및 `complex()`는 *__index__()*를 사용합니다.

object.**__round__**(*self*[, *ndigits*])

object.**__trunc__**(*self*)

object.**__floor__**(*self*)

object.**__ceil__**(*self*)

내장 함수 `round()`와 `math` 함수 `trunc()`, `floor()`, `ceil()`을 구현하기 위해 호출됩니다. *ndigits*가 `__round__()`로 전달되지 않는 한, 이 메서드들은 모두 `Integral`(보통 `int`)로 잘린 객체의 값을 돌려줘야 합니다.

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

3.3.9 with 문 컨텍스트 관리자

컨텍스트 관리자(*context manager*)는 *with* 문을 실행할 때 자리 잡는 실행 컨텍스트(context)를 정의하는 객체입니다. 코드 블록의 실행을 위해, 컨텍스트 관리자는 원하는 실행시간 컨텍스트로의 진입과 탈출을 처리합니다. 컨텍스트 관리자는 보통 `with` 문(*with* 문 섹션에서 설명합니다)으로 시작되지만, 그들의 메서드를 호출해서 직접 사용할 수도 있습니다.

컨텍스트 관리자의 전형적인 용도에는 다양한 종류의 전역 상태(global state)를 보관하고 복구하는 것, 자원을 로킹(locking)하고 언로킹(unlocking)하는 것, 열린 파일을 닫는 것 등이 있습니다.

For more information on context managers, see typecontextmanager. The `object` class itself does not provide the context manager methods.

`object.`**`__enter__`**(*self*)

이 객체와 연관된 실행시간 컨텍스트에 진입합니다. *`with`* 문은 `as` 절로 지정된 대상이 있다면, 이 메서드의 반환 값을 연결합니다.

`object.`**`__exit__`**(*self*, *exc_type*, *exc_value*, *traceback*)

이 객체와 연관된 실행시간 컨텍스트를 종료합니다. 매개변수들은 컨텍스트에서 벗어나게 만든 예외를 기술합니다. 만약 컨텍스트가 예외 없이 종료한다면, 세 인자 모두 `None` 이 됩니다.

만약 예외가 제공되고, 메서드가 예외를 중지시키고 싶으면 (즉 확산하는 것을 막으려면) 참(true)을 돌려줘야 합니다. 그렇지 않으면 예외는 이 메서드가 종료한 후에 계속 진행됩니다.

Note that *`__exit__()`* methods should not reraise the passed-in exception; this is the caller' s responsibility.

더 보기

PEP 343 - "with" 문
파이썬 *`with`* 문에 대한 규격, 배경, 예.

3.3.10 Customizing positional arguments in class pattern matching

When using a class name in a pattern, positional arguments in the pattern are not allowed by default, i.e. `case MyClass(x, y)` is typically invalid without special support in `MyClass`. To be able to use that kind of pattern, the class needs to define a *__match_args__* attribute.

`object.`**`__match_args__`**

This class variable can be assigned a tuple of strings. When this class is used in a class pattern with positional arguments, each positional argument will be converted into a keyword argument, using the corresponding value in *__match_args__* as the keyword. The absence of this attribute is equivalent to setting it to `()`.

For example, if `MyClass.__match_args__` is `("left", "center", "right")` that means that `case MyClass(x, y)` is equivalent to `case MyClass(left=x, center=y)`. Note that the number of arguments in the pattern must be smaller than or equal to the number of elements in *__match_args__*; if it is larger, the pattern match attempt will raise a `TypeError`.

Added in version 3.10.

더 보기

PEP 634 - Structural Pattern Matching
The specification for the Python `match` statement.

3.3.11 Emulating buffer types

The buffer protocol provides a way for Python objects to expose efficient access to a low-level memory array. This protocol is implemented by builtin types such as `bytes` and `memoryview`, and third-party libraries may define additional buffer types.

While buffer types are usually implemented in C, it is also possible to implement the protocol in Python.

`object.`**`__buffer__`**(*self*, *flags*)

Called when a buffer is requested from *self* (for example, by the `memoryview` constructor). The *flags* argument is an integer representing the kind of buffer requested, affecting for example whether the returned buffer is read-only or writable. `inspect.BufferFlags` provides a convenient way to interpret the flags. The method must return a `memoryview` object.

object.**__release_buffer__**(*self*, *buffer*)

Called when a buffer is no longer needed. The *buffer* argument is a `memoryview` object that was previously returned by `__buffer__()`. The method must release any resources associated with the buffer. This method should return `None`. Buffer objects that do not need to perform any cleanup are not required to implement this method.

Added in version 3.12.

더 보기

PEP 688 - Making the buffer protocol accessible in Python
Introduces the Python `__buffer__` and `__release_buffer__` methods.

`collections.abc.Buffer`
ABC for buffer types.

3.3.12 Annotations

Functions, classes, and modules may contain *annotations*, which are a way to associate information (usually *type hints*) with a symbol.

object.**__annotations__**

This attribute contains the annotations for an object. It is *lazily evaluated*, so accessing the attribute may execute arbitrary code and raise exceptions. If evaluation is successful, the attribute is set to a dictionary mapping from variable names to annotations.

버전 3.14에서 변경: Annotations are now lazily evaluated.

object.**__annotate__**(*format*)

An *annotate function*. Returns a new dictionary object mapping attribute/parameter names to their annotation values.

Takes a format parameter specifying the format in which annotations values should be provided. It must be a member of the `annotationlib.Format` enum, or an integer with a value corresponding to a member of the enum.

If an annotate function doesn't support the requested format, it must raise `NotImplementedError`. Annotate functions must always support `VALUE` format; they must not raise `NotImplementedError()` when called with this format.

When called with `VALUE` format, an annotate function may raise `NameError`; it must not raise `NameError` when called requesting any other format.

If an object does not have any annotations, `__annotate__` should preferably be set to `None` (it can't be deleted), rather than set to a function that returns an empty dict.

Added in version 3.14.

더 보기

PEP 649 — Deferred evaluation of annotation using descriptors
Introduces lazy evaluation of annotations and the `__annotate__` function.

3.3.13 특수 메서드 조회

사용자 정의 클래스의 경우, 묵시적인 특수 메서드의 호출은 객체의 인스턴스 딕셔너리가 아닌 객체의 형에 정의되어 있을 때만 올바르게 동작함이 보장됩니다. 이런 동작은 다음과 같은 코드가 예외를 일으키는 원인입니다:

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

The rationale behind this behaviour lies with a number of special methods such as *__hash__()* and *__repr__()* that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when invoked on the type object itself:

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

클래스의 연결되지 않은 메서드를 호출하려는 이런 식의 잘못된 시도는 종종 '메타 클래스 혼란(metaclass confusion)' 이라고 불리고, 특수 메서드를 조회할 때 인스턴스를 우회하는 방법으로 피할 수 있습니다.

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the *__getattribute__()* method even of the object' s metaclass:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print("Class getattribute invoked")
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
10
>>> len(c)                      # Implicit lookup
10
```

Bypassing the *__getattribute__()* machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method *must* be set on the class object itself in order to be consistently invoked by the interpreter).

3.4 코루틴(Coroutines)

3.4.1 어웨이터블 객체(Awaitable Objects)

An *awaitable* object generally implements an `__await__()` method. *Coroutine objects* returned from `async def` functions are awaitable.

> **참고**
>
> The *generator iterator* objects returned from generators decorated with `types.coroutine()` are also awaitable, but they do not implement `__await__()`.

`object.`**`__await__`**(*self*)

Must return an *iterator*. Should be used to implement *awaitable* objects. For instance, `asyncio.Future` implements this method to be compatible with the `await` expression. The `object` class itself is not awaitable and does not provide this method.

> **참고**
>
> The language doesn' t place any restriction on the type or value of the objects yielded by the iterator returned by `__await__`, as this is specific to the implementation of the asynchronous execution framework (e.g. `asyncio`) that will be managing the *awaitable* object.

Added in version 3.5.

> **더 보기**
>
> **PEP 492** 가 어웨이터블 객체에 대한 더 자세한 정보를 포함하고 있습니다.

3.4.2 코루틴 객체(Coroutine Objects)

Coroutine objects are *awaitable* objects. A coroutine' s execution can be controlled by calling `__await__()` and iterating over the result. When the coroutine has finished executing and returns, the iterator raises `StopIteration`, and the exception' s `value` attribute holds the return value. If the coroutine raises an exception, it is propagated by the iterator. Coroutines should not directly raise unhandled `StopIteration` exceptions.

코루틴은 다음에 나열하는 메서드들 또한 갖고 있는데, 제너레이터(제너레이터-이터레이터 메서드 를 보십시오)의 것들과 닮았습니다. 하지만, 제너레이터와는 달리, 코루틴은 이터레이션을 직접 지원하지는 않습니다.

버전 3.5.2에서 변경: 코루틴을 두 번 await 하면 `RuntimeError` 를 일으킵니다.

`coroutine.`**`send`**(*value*)

Starts or resumes execution of the coroutine. If *value* is `None`, this is equivalent to advancing the iterator returned by `__await__()`. If *value* is not `None`, this method delegates to the `send()` method of the iterator that caused the coroutine to suspend. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above.

`coroutine.`**`throw`**(*value*)

`coroutine.`**`throw`**(*type*[, *value*[, *traceback*]])

Raises the specified exception in the coroutine. This method delegates to the `throw()` method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.

버전 3.12에서 변경: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

coroutine.**close**()

코루틴이 자신을 정리하고 종료하도록 만듭니다. 만약 코루틴이 일시 중지 중이면, 이 메서드는 먼저 코루틴이 일시 중지되도록 한 이터레이터의 *close()* 메서드로 위임합니다(그런 메서드를 가지는 경우). 그런 다음 일시 중지지점에서 GeneratorExit 를 발생시키는데, 코루틴이 즉시 자신을 정리하도록 만듭니다. 마지막으로 코루틴에 실행을 종료했다고 표시하는데, 아직 시작하지조차 않았을 때도 그렇다.

코루틴 객체가 파괴될 때는 위의 프로세스에 따라 자동으로 닫힙니다(closed).

3.4.3 비동기 이터레이터(Asynchronous Iterators)

비동기 이터레이터 는 자신의 __anext__ 메서드에서 비동기 코드를 호출할 수 있습니다.

비동기 이터레이터는 *async for* 문에서 사용될 수 있습니다.

The object class itself does not provide these methods.

object.**__aiter__**(*self*)

비동기 이터레이터 객체를 돌려줘야 합니다.

object.**__anext__**(*self*)

이터레이터의 다음 값을 주는 어웨이터블 을 돌려줘야 합니다. 이터레이션이 끝나면 StopAsyncIteration 에러를 일으켜야 합니다.

비동기 이터러블 객체의 예:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Added in version 3.5.

버전 3.7에서 변경: Prior to Python 3.7, *__aiter__()* could return an *awaitable* that would resolve to an *asynchronous iterator*.

Starting with Python 3.7, *__aiter__()* must return an asynchronous iterator object. Returning anything else will result in a TypeError error.

3.4.4 비동기 컨텍스트 관리자

비동기 컨텍스트 관리자(*asynchronous context manager*) 는 __aenter__ 와 __aexit__ 메서드에서 실행을 일시 중지할 수 있는 컨텍스트 관리자 입니다.

비동기 컨텍스트 관리자는 *async with* 문에서 사용될 수 있습니다.

The object class itself does not provide these methods.

object.**__aenter__**(*self*)

Semantically similar to *__enter__()*, the only difference being that it must return an *awaitable*.

object.**__aexit__**(*self*, *exc_type*, *exc_value*, *traceback*)

Semantically similar to *__exit__()*, the only difference being that it must return an *awaitable*.

비동기 컨텍스트 관리자 클래스의 예:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Added in version 3.5.

CHAPTER 4

실행 모델

4.1 프로그램의 구조

파이썬 프로그램은 코드 블록으로 만들어집니다. 블록 (*block*) 은 한 단위로 실행되는 한 조각의 파이썬 프로그램 텍스트입니다. 다음과 같은 것들이 블록입니다: 모듈, 함수 바디, 클래스 정의. 대화형으로 입력되는 각 명령은 블록입니다. 스크립트 파일(표준 입력을 통해 인터프리터로 제공되는 파일이나 인터프리터에 명령행 인자로 지정된 파일)은 코드 블록입니다. 스크립트 명령(`-c` 옵션으로 인터프리터 명령행에 지정된 명령)은 코드 블록입니다. `-m` 인자를 사용하여 명령 줄에서 최상위 수준 스크립트로 (모듈 `__main__`으로) 실행되는 모듈도 코드 블록입니다. 내장함수 `eval()` 과 `exec()` 로 전달되는 문자열 인자도 코드 블록입니다.

코드 블록은 실행 프레임 (*execution frame*) 에서 실행됩니다. 프레임은 몇몇 관리를 위한 정보(디버깅에 사용됩니다)를 포함하고, 코드 블록의 실행이 끝난 후에 어디서 어떻게 실행을 계속할 것인지를 결정합니다.

4.2 이름과 연결(binding)

4.2.1 이름의 연결

이름 (*Names*) 은 객체를 가리킵니다. 이름은 이름 연결 연산 때문에 만들어집니다.

The following constructs bind names:

- formal parameters to functions,
- class definitions,
- function definitions,
- assignment expressions,
- *targets* that are identifiers if occurring in an assignment:
  - *for* loop header,
  - after `as` in a *with* statement, *except* clause, *except** clause, or in the as-pattern in structural pattern matching,
  - in a capture pattern in structural pattern matching
- *import* statements.
- *type* statements.

- *type parameter lists*.

The `import` statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

`del` 문에 나오는 대상 역시 이 목적에서 연결된 것으로 간주합니다(실제 의미가 이름을 연결 해제하는 것이기는 해도).

각 대입이나 임포트 문은 클래스나 함수 정의 때문에 정의되는 블록 내에 등장할 수 있고, 모듈 수준(최상위 코드 블록)에서 등장할 수도 있습니다.

If a name is bound in a block, it is a local variable of that block, unless declared as `nonlocal` or `global`. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a *free variable*.

프로그램 텍스트에 등장하는 각각의 이름들은 다음에 나오는 이름 검색(name resolution) 규칙에 따라 확정되는 이름의 연결 (*binding*) 을 가리킵니다.

4.2.2 이름의 검색(resolution)

스코프 (*scope*) 는 블록 내에서 이름의 가시성(visibility)을 정의합니다. 지역 변수가 블록에서 정의되면, 그것의 스코프는 그 블록을 포함합니다. 만약 정의가 함수 블록에서 이루어지면, 포함된 블록이 그 이름에 대해 다른 결합을 만들지 않는 이상, 스코프는 정의하고 있는 것 안에 포함된 모든 블록으로 확대됩니다.

이름이 코드 블록 내에서 사용될 때, 가장 가깝게 둘러싸고 있는 스코프에 있는 것으로 검색됩니다. 코드 블록이 볼 수 있는 모든 스코프의 집합을 블록의 환경 (*environment*) 이라고 부릅니다.

이름이 어디에서도 발견되지 않으면 `NameError` 예외가 발생합니다. 만약 현재 스코프가 함수 스코프이고, 그 이름이 사용되는 시점에 아직 연결되지 않은 지역 변수면 `UnboundLocalError` 예외가 발생합니다. `UnboundLocalError` 는 `NameError` 의 서브 클래스입니다.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations. See the FAQ entry on UnboundLocalError for examples.

If the `global` statement occurs within a block, all uses of the names specified in the statement refer to the bindings of those names in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module `builtins`. The global namespace is searched first. If the names are not found there, the builtins namespace is searched next. If the names are also not found in the builtins namespace, new variables are created in the global namespace. The global statement must precede all uses of the listed names.

`global` 문은 같은 블록의 이름 연결 연산과 같은 스코프를 갖습니다. 자유 변수의 경우 가장 가까이서 둘러싸는 스코프가 global 문을 포함한다면, 그 자유 변수는 전역으로 취급됩니다.

The `nonlocal` statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. `SyntaxError` is raised at compile time if the given name does not exist in any enclosing function scope. *Type parameters* cannot be rebound with the `nonlocal` statement.

모듈의 이름 공간은 모듈이 처음 임포트될 때 자동으로 만들어집니다. 스크립트의 메인 모듈은 항상 `__main__` 이라고 불립니다.

Class definition blocks and arguments to `exec()` and `eval()` are special in the context of name resolution. A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution with an exception that unbound local variables are looked up in the global namespace. The namespace of the class definition becomes the attribute dictionary of the class. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods. This includes comprehensions and generator expressions, but it does not include *annotation scopes*, which have access to their enclosing class scopes. This means that the following will fail:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

However, the following will succeed:

```
class A:
    type Alias = Nested
    class Nested: pass

print(A.Alias.__value__)  # <type 'A.Nested'>
```

4.2.3 Annotation scopes

Annotations, *type parameter lists* and `type` statements introduce *annotation scopes*, which behave mostly like function scopes, but with some exceptions discussed below.

Annotation scopes are used in the following contexts:

- *Function annotations*.
- *Variable annotations*.
- Type parameter lists for *generic type aliases*.
- Type parameter lists for *generic functions*. A generic function' s annotations are executed within the annotation scope, but its defaults and decorators are not.
- Type parameter lists for *generic classes*. A generic class' s base classes and keyword arguments are executed within the annotation scope, but its decorators are not.
- The bounds, constraints, and default values for type parameters (*lazily evaluated*).
- The value of type aliases (*lazily evaluated*).

Annotation scopes differ from function scopes in the following ways:

- Annotation scopes have access to their enclosing class namespace. If an annotation scope is immediately within a class scope, or within another annotation scope that is immediately within a class scope, the code in the annotation scope can use names defined in the class scope as if it were executed directly within the class body. This contrasts with regular functions defined within classes, which cannot access names defined in the class scope.
- Expressions in annotation scopes cannot contain `yield`, `yield from`, `await`, or `:=` expressions. (These expressions are allowed in other scopes contained within the annotation scope.)
- Names defined in annotation scopes cannot be rebound with `nonlocal` statements in inner scopes. This includes only type parameters, as no other syntactic elements that can appear within annotation scopes can introduce new names.
- While annotation scopes have an internal name, that name is not reflected in the *qualified name* of objects defined within the scope. Instead, the `__qualname__` of such objects is as if the object were defined in the enclosing scope.

Added in version 3.12: Annotation scopes were introduced in Python 3.12 as part of **PEP 695**.

버전 3.13에서 변경: Annotation scopes are also used for type parameter defaults, as introduced by **PEP 696**.

버전 3.14에서 변경: Annotation scopes are now also used for annotations, as specified in **PEP 649** and **PEP 749**.

4.2.4 Lazy evaluation

Most annotation scopes are *lazily evaluated*. This includes annotations, the values of type aliases created through the `type` statement, and the bounds, constraints, and default values of type variables created through the *type parameter syntax*. This means that they are not evaluated when the type alias or type variable is created, or when the object

carrying annotations is created. Instead, they are only evaluated when necessary, for example when the `__value__` attribute on a type alias is accessed.

Example:

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def func[T: 1/0](): pass
>>> T = func.__type_params__[0]
>>> T.__bound__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

Here the exception is raised only when the `__value__` attribute of the type alias or the `__bound__` attribute of the type variable is accessed.

This behavior is primarily useful for references to types that have not yet been defined when the type alias or type variable is created. For example, lazy evaluation enables creation of mutually recursive type aliases:

```
from typing import Literal

type SimpleExpr = int | Parenthesized
type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]
type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"], Expr]
```

Lazily evaluated values are evaluated in *annotation scope*, which means that names that appear inside the lazily evaluated value are looked up as if they were used in the immediately enclosing scope.

Added in version 3.12.

4.2.5 builtins 와 제한된 실행

사용자는 `__builtins__` 를 건드리지 말아야 합니다; 이것은 구현 세부사항입니다. 내장 이름 공간의 값을 변경하고 싶은 사용자는 `builtins` 모듈을 *import* 하고 그것의 어트리뷰트를 적절하게 수정해야 합니다.

코드 블록의 실행과 연관된 내장 이름 공간은, 사실 전역 이름 공간의 이름 `__builtins__` 를 조회함으로써 발견됩니다. 이것은 딕셔너리나 모듈이어야 합니다(후자의 경우 모듈의 딕셔너리가 사용됩니다). 기본적으로, `__main__` 모듈에 있을 때는 `__builtins__` 가 내장 모듈 `builtins` 이고, 다른 모듈에 있을 때는 `__builtins__` 는 `builtins` 모듈의 딕셔너리에 대한 별칭입니다.

4.2.6 동적 기능과의 상호작용

자유 변수에 대해 이름 검색은 컴파일 시점이 아니라 실행 시점에 이루어집니다. 이것은 다음과 같은 코드가 42를 출력한다는 것을 뜻합니다:

```
i = 10
def f():
    print(i)
i = 42
f()
```

`eval()` 과 `exec()` 함수는 이름 검색을 위한 완전한 환경에 대한 접근권이 없습니다. 이름은 호출자의 지역과 전역 이름 공간에서 검색될 수 있습니다. 자유 변수는 가장 가까이 둘러싼 이름 공간이 아니라 전역 이름 공간에서 검색됩니다.[1] `exec()` 과 `eval()` 함수에는 전역과 지역 이름 공간을 재정의할 수 있는 생략 가능한 인자가 있습니다. 만약 단지 한 이름 공간만 주어지면, 그것이 두 가지 모두로 사용됩니다.

[1] 이 한계는 이 연산들 때문에 실행되는 코드가 모듈이 컴파일되는 시점에는 존재하지 않았기 때문입니다.

4.3 예외

예외는 에러나 예외적인 조건을 처리하기 위해 코드 블록의 일반적인 제어 흐름을 깨는 수단입니다. 에러가 감지된 지점에서 예외를 일으킵니다(*raised*); 둘러싼 코드 블록이나 직접적 혹은 간접적으로 에러가 발생한 코드 블록을 호출한 어떤 코드 블록에서건 예외는 처리될 수 있습니다.

파이썬 인터프리터는 실행 시간 에러(0으로 나누는 것 같은)를 감지할 때 예외를 일으킵니다. 파이썬 프로그램은 `raise` 문을 사용해서 명시적으로 예외를 일으킬 수 있습니다. 예외 처리기는 `try` … `except` 문으로 지정됩니다. 그런 문장에서 `finally` 구는 정리(cleanup) 코드를 지정하는 데 사용되는데, 예외를 처리하는 것이 아니라 앞선 코드에서 예외가 발생하건 그렇지 않건 실행됩니다.

파이썬은 에러 처리에 "종결 (termination)" 모델을 사용합니다; 예외 처리기가 뭐가 발생했는지 발견할 수 있고, 바깥 단계에서 실행을 계속할 수는 있지만, 에러의 원인을 제거한 후에 실패한 연산을 재시도할 수는 없습니다(문제의 코드 조각을 처음부터 다시 시작시키는 것은 예외입니다).

예외가 어디서도 처리되지 않을 때, 인터프리터는 프로그램의 실행을 종료하거나, 대화형 메인 루프로 돌아갑니다. 두 경우 모두, 예외가 `SystemExit` 인 경우를 제외하고, 스택 트레이스백을 인쇄합니다.

Exceptions are identified by class instances. The `except` clause is selected depending on the class of the instance: it must reference the class of the instance or a *non-virtual base class* thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.

> **참고**
>
> 예외 메시지는 파이썬 API 일부가 아닙니다. 그 내용은 파이썬의 버전이 바뀔 때 경고 없이 변경될 수 있고, 코드는 여러 버전의 인터프리터에서 실행될 수 있는 코드는 이것에 의존하지 말아야 합니다.

섹션 *try 문* 에서 `try` 문, *raise 문* 에서 `raise` 문에 대한 설명이 제공됩니다.

CHAPTER 5

임포트 시스템

한 모듈 에 있는 파이썬 코드는 임포팅 이라는 프로세스를 통해 다른 모듈에 있는 코드들에 대한 접근권을 얻습니다. *import* 문은 임포트 절차를 일으키는 가장 흔한 방법이지만, 유일한 방법은 아닙니다. `importlib.import_module()` 같은 함수나 내장 `__import__()` 도 임포트 절차를 일으키는데 사용될 수 있습니다.

import 문은 두 가지 연산을 합친 것입니다; 먼저 이름이 가리키는 모듈을 찾은 후에, 그 검색의 결과를 지역 스코프의 이름에 연결합니다. `import` 문의 검색 연산은 적절한 인자들로 `__import__()` 함수를 호출하는 것으로 정의됩니다. `__import__()` 의 반환 값은 `import` 문의 이름 연결 연산을 수행하는 데 사용됩니다. 이 이름 연결 연산의 정확한 세부사항에 대해서는 `import` 문을 보세요.

`__import__()` 의 직접 호출은 모듈을 찾고, 발견된다면, 모듈을 만드는 연산만을 수행합니다. 부모 패키지를 임포트하거나 여러 캐시(`sys.modules` 를 포함합니다)를 갱신하는 것과 같은 부수적인 효과들이 일어날 수 있기는 하지만, 오직 *import* 문만이 이름 연결 연산을 수행합니다.

import 문이 실행될 때, 표준 내장 `__import__()` 가 호출됩니다. 임포트 시스템을 호출하는 다른 메커니즘(`importlib.import_module()` 같은)은 `__import__()` 를 사용하지 않고 임포트 개념을 구현하기 위한 자신의 방법을 사용할 수 있습니다.

모듈이 처음 임포트 될 때, 파이썬은 모듈을 검색하고, 발견된다면, 모듈 객체를 만들고[1], 초기화합니다. 만약 그 이름의 모듈을 발견할 수 없다면, `ModuleNotFoundError` 를 일으킵니다. 파이썬은 임포트 절차가 호출될 때 이름 붙여진 모듈을 찾는 다양한 전략을 구현합니다. 이 전략들은 다음 섹션에서 설명하는 여러 가지 훅을 통해 수정되고 확장될 수 있습니다.

버전 3.3에서 변경: 임포트 시스템은 **PEP 302** 의 두 번째 단계를 완전히 구현하도록 개정되었습니다. 이제 묵시적인 임포트 절차는 없습니다 - 전체 임포트 시스템이 `sys.meta_path` 을 통해 노출됩니다. 여기에 더해, 네이티브(native) 이름 공간 패키지의 지원이 구현되었습니다 (**PEP 420** 을 보세요).

5.1 importlib

`importlib` 모듈은 임포트 시스템과 상호 작용하기 위한 풍부한 API를 제공합니다. 예를 들어, `importlib.import_module()` 는 임포트 절차를 구동하는 데 있어 내장 `__import__()` 에 비해 권장되고, 더 간단한 API를 제공합니다. 더 상세한 내용은 `importlib` 라이브러리 설명서를 참조하십시오.

[1] `types.ModuleType` 을 보세요.

5.2 패키지(package)

파이썬은 한 가지 종류의 모듈 객체만 갖고 있고, 모든 모듈은 모듈이 파이썬이나 C나 그 밖의 다른 어떤 방법으로 구현되었는지와 상관없이 이 형입니다. 모듈을 조직화하고 이름 계층구조를 제공하기 위해, 파이썬은 패키지 라는 개념을 갖고 있습니다.

패키지를 파일 시스템에 있는 디렉터리라고 생각할 수 있지만, 패키지와 모듈이 파일시스템으로부터 올 필요는 없으므로 이 비유를 너무 문자 그대로 해석하지 말아야 합니다. 이 문서의 목적상, 디렉터리와 파일이라는 비유를 사용할 것입니다. 파일 시스템 디렉터리처럼, 패키지는 계층적으로 조직화하고, 패키지는 보통 모듈뿐만 아니라 서브 패키지도 포함할 수 있습니다.

모든 패키지가 모듈이라는 것을 기억하는 것이 중요합니다. 하지만 모든 모듈이 패키지인 것은 아닙니다. 다른 식으로 표현하면, 패키지는 특별한 종류의 모듈입니다. 구체적으로, `__path__` 어트리뷰트를 포함하는 모든 모듈은 패키지로 취급됩니다.

All modules have a name. Subpackage names are separated from their parent package name by a dot, akin to Python's standard attribute access syntax. Thus you might have a package called `email`, which in turn has a subpackage called `email.mime` and a module within that subpackage called `email.mime.text`.

5.2.1 정규 패키지

파이썬은 두 가지 종류의 패키지를 정의합니다, 정규 패키지 와 이름 공간 패키지. 정규 패키지는 파이썬 3.2와 그 이전에 존재하던 전통적인 패키지입니다. 정규 패키지는 보통 `__init__.py` 파일을 가진 디렉터리로 구현됩니다. 정규 패키지가 임포트될 때, 이 `__init__.py` 파일이 묵시적으로 실행되고, 그것이 정의하는 객체들이 패키지의 이름 공간의 이름들도 연결됩니다. `__init__.py` 파일은 다른 모듈들이 가질 수 있는 것과 같은 파이썬 코드를 포함할 수 있고, 파이썬은 임포트될 때 모듈에 몇 가지 어트리뷰트를 추가합니다.

예를 들어, 다음과 같은 파일시스템 배치는 최상위 `parent` 패키지와 세 개의 서브 패키지를 정의합니다:

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

`parent.one` 을 임포트하면 `parent/__init__.py` 과 `parent/one/__init__.py` 을 묵시적으로 실행합니다. 뒤이은 `parent.two` 와 `parent.three` 의 임포트는 각각 `parent/two/__init__.py` 와 `parent/three/__init__.py` 를 실행합니다.

5.2.2 이름 공간 패키지

이름 공간 패키지는 여러 가지 포션 들의 복합체인데, 각 포션들은 부모 패키지의 서브 패키지로 이바지합니다. 포션들은 파일시스템의 다른 위치에 놓일 수 있습니다. 포션들은 zip 파일이나 네트워크나 파이썬이 임포트 할 때 검색하는 어떤 다른 장소에서 발견될 수 있습니다. 이름 공간 패키지는 파일시스템의 객체와 직접적인 상관관계가 있을 수도 있고 그렇지 않을 수도 있습니다; 구체적인 형태가 없는 가상 모듈일 수도 있습니다.

이름 공간 패키지는 `__path__` 어트리뷰트로 일반적인 리스트를 사용하지 않습니다. 대신에 특별한 이터러블 형을 사용하는데, 그 패키지 내의 다음 임포트 시도에서 그것의 부모 패키지(또는 최상위 패키지의 경우 `sys.path`) 의 경로가 변했으면 패키지 포션에 대한 새 검색을 자동으로 수행하게 됩니다.

이름 공간 패키지의 경우, `parent/__init__.py` 파일이 없습니다. 사실, 임포트 검색 동안 여러 개의 `parent` 디렉터리가 발견될 수 있고, 각각의 것은 다른 포션들에 의해 제공됩니다. 그래서 `parent/one` 은 물리적으로 `parent/two` 옆에 위치하지 않을 수 있습니다. 이 경우, 파이썬은 자신 또는 서브 패키지 중 어느 하나가 임포트 될 때마다 최상위 `parent` 패키지를 위한 이름 공간 패키지를 만듭니다.

이름 공간 패키지의 규격은 **PEP 420** 을 참조하세요.

5.3 검색

검색을 시작하기 위해, 파이썬은 임포트될 모듈(또는 패키지, 하지만 이 논의에서 차이점은 중요하지 않다)의 완전히 정규화된 이름 을 필요로 합니다. 이 이름은 *import* 문으로 제공된 여러 인자나, `importlib.import_module()` 나 `__import__()` 함수로 전달된 매개변수들로부터 옵니다.

이 이름은 임포트 검색의 여러 단계에서 사용되는데, 서브 모듈로 가는 점으로 구분된 경로일 수 있습니다, 예를 들어 `foo.bar.baz`. 이 경우에, 파이썬은 먼저 `foo` 를, 그다음에 `foo.bar` 를, 마지막으로 `foo.bar.baz` 를 임포트하려고 시도합니다. 만약 중간 임포트가 어느 하나라도 실패한다면 `ModuleNotFoundError` 가 발생합니다.

5.3.1 모듈 캐시

임포트 검색 도중 처음으로 검사되는 장소는 `sys.modules` 입니다. 이 매핑은 중간 경로들을 포함해서 전에 임포트 된 모든 모듈의 캐시로 기능합니다. 그래서 만약 `foo.bar.baz` 가 앞서 임포트 되었다면, `sys.modules` 는 `foo`, `foo.bar`, `foo.bar.baz` 항목들을 포함합니다. 각 키에 대응하는 값들은 모듈 객체입니다.

임포트하는 동안, 모듈 이름을 `sys.modules` 에서 찾고, 만약 있다면 해당 값이 임포트를 만족하는 모듈이고, 프로세스는 완료됩니다. 하지만 값이 `None` 이면, `ModuleNotFoundError` 를 일으킵니다. 만약 모듈 이름이 없다면, 파이썬은 모듈 검색을 계속 진행합니다.

`sys.modules` 은 쓰기가 허락됩니다. 키를 삭제해도 해당 모듈을 파괴하지는 않지만(다른 모듈들이 아직 그 모듈에 대한 참조를 유지하고 있을 수 있으므로), 해당 이름의 모듈에 대한 캐시를 무효화해서, 다음 임포트때 파이썬으로 하여금 그 모듈을 다시 찾도록 만듭니다. 키에는 `None` 을 대입할 수도 있는데, 다음 임포트 때 `ModuleNotFoundError` 가 일어나도록 만듭니다.

모듈 객체에 대한 참조를 유지한다면, `sys.modules` 의 캐시 항목을 무효로 한 후 다시 임포트하면 두 모듈 객체는 같은 것이 아니게 됨에 주의해야 합니다. 반면에 `importlib.reload()` 는 같은 모듈 객체를 재사용하고, 간단하게 모듈의 코드를 다시 실행해서 모듈의 내용을 다시 초기화합니다.

5.3.2 파인더(finder)와 로더(loader)

모듈이 `sys.modules` 에서 발견되지 않으면, 모듈을 찾아서 로드하기 위해 파이썬의 임포트 프로토콜이 구동됩니다. 이 프로토콜은 두 개의 개념적 객체들로 구성되어 있습니다, 파인더 와 로더. 파인더의 일은 자신이 알고 있는 전략을 사용해, 주어진 이름의 모듈을 찾을 수 있는지 결정하는 것입니다. 두 인터페이스 모두를 구현한 객체들을 임포터라고 부릅니다 - 요청한 모듈을 로딩할 수 있다고 판단할 때 자신을 돌려줍니다.

파이썬은 여러 가지 기본 파인더들과 임포터들을 포함하고 있습니다. 첫 번째 것은 내장 모듈들의 위치를 찾을 수 있고, 두 번째 것은 프로즌 모듈(frozen module)의 위치를 찾을 수 있고, 세 번째 것은 모듈을 임포트 경로 에서 검색합니다. 임포트 경로 는 파일 시스템의 경로나 zip 파일을 가리키는 위치들의 목록입니다. 그것은 URL로 식별될 수 있는 것들처럼, 위치가 지정될 수 있는 자원들을 검색하도록 확장될 수 있습니다.

임포트 절차는 확장 가능해서, 모듈 검색의 범위를 확대하기 위해 새 파인더를 추가할 수 있습니다.

파인더는 실제로 모듈을 로드하지는 않습니다. 주어진 이름의 모듈을 찾으면 임포트와 관련된 정보들을 요약한 모듈 스펙 (*module spec*) 을 돌려주는데, 임포트 절차는 모듈을 로딩할 때 이것을 사용하게 됩니다.

다음 섹션은 파인더와 로더의 프로토콜에 대해 좀 더 자세히 설명하는데, 임포트 절차를 확장하기 위해 어떻게 새로운 것들을 만들고 등록하는지를 포함합니다.

버전 3.4에서 변경: 이전 버전의 파이썬에서, 파인더가 로더 를 직접 돌려주었지만, 이제는 로더를 포함하고 있는 모듈 스펙을 돌려줍니다. 임포트 도중 로더가 아직 사용되기는 하지만 그 역할은 축소되었습니다.

5.3.3 임포트 훅(import hooks)

임포트 절차는 확장할 수 있도록 설계되었습니다; 일차적인 메커니즘은 임포트 훅(*import hook*) 입니다. 두 가지 종류의 임포트 훅이 있습니다: 메타 훅(*meta hook*) 과 임포트 경로 훅(*import path hook*).

메타 훅은 임포트 처리의 처음에, `sys.modules` 캐시 조회를 제외한 다른 임포트 처리들이 시작되기 전에 호출됩니다. 이것은 메타 훅이 `sys.path` 처리, 프로즌 모듈, 내장 모듈들을 재정의할 수 있게 합니다. 다음에 설명하듯이, 메타 훅은 `sys.meta_path` 에 새 파인더 객체를 추가하는 방법으로 등록할 수 있습니다.

임포트 경로 혹은 sys.path (혹은 package.__path__) 처리 일부로, 관련된 경로 항목을 만나는 시점에 호출됩니다. 다음에 설명하듯이, 임포트 경로 혹은 sys.path_hooks 에 새 콜러블을 추가하는 방법으로 등록할 수 있습니다.

5.3.4 메타 경로(meta path)

When the named module is not found in sys.modules, Python next searches sys.meta_path, which contains a list of meta path finder objects. These finders are queried in order to see if they know how to handle the named module. Meta path finders must implement a method called find_spec() which takes three arguments: a name, an import path, and (optionally) a target module. The meta path finder can use any strategy it wants to determine whether it can handle the named module or not.

만약 메타 경로 파인더가 주어진 이름의 모듈을 처리하는 법을 안다면, 스펙 객체를 돌려줍니다. 그럴 수 없다면 None 을 돌려줍니다. 만약 sys.meta_path 처리가 스펙을 돌려주지 못하고 목록의 끝에 도달하면, ModuleNotFoundError 를 일으킵니다. 발생하는 다른 예외들은 그냥 확산시키고, 임포트 프로세스를 중단합니다.

The find_spec() method of meta path finders is called with two or three arguments. The first is the fully qualified name of the module being imported, for example foo.bar.baz. The second argument is the path entries to use for the module search. For top-level modules, the second argument is None, but for submodules or subpackages, the second argument is the value of the parent package' s __path__ attribute. If the appropriate __path__ attribute cannot be accessed, a ModuleNotFoundError is raised. The third argument is an existing module object that will be the target of loading later. The import system passes in a target module only during reload.

메타 경로는 한 번의 임포트 요청에 대해 여러 번 탐색 될 수 있습니다. 예를 들어, 대상 모듈들이 아무것도 캐싱 되지 않았다고 할 때, foo.bar.baz 를 임포트 하려면, 먼저 각 메타 경로 파인더 (mpf)들에 대해 mpf.find_spec("foo", None, None) 를 호출해서 최상위 임포트를 수행합니다. foo 가 임포트 된 후에, 메타 경로를 두 번째 탐색해서 foo.bar 를 임포트 하는데, mpf.find_spec("foo.bar", foo.__path__, None) 를 호출합니다. 일단 foo.bar 가 임포트 되면, 마지막 탐색은 mpf.find_spec("foo.bar.baz", foo.bar.__path__, None) 를 호출합니다.

어떤 메타 경로 파인더들은 오직 최상위 임포트만 지원합니다. 이런 임포터들은 두 번째 인자로 None 이 아닌 것이 오면 항상 None 을 돌려줍니다.

파이썬의 기본 sys.meta_path 는 세 개의 메타 경로 파인더를 갖고 있습니다. 하나는 내장 모듈을 임포트 하는 법을 알고, 하나는 프로즌 모듈을 임포트하는 법을 알고, 하나는 임포트 경로 에서 모듈을 임포트하는 법을 압니다(즉 경로 기반 파인더).

버전 3.4에서 변경: The find_spec() method of meta path finders replaced find_module(), which is now deprecated. While it will continue to work without change, the import machinery will try it only if the finder does not implement find_spec().

버전 3.10에서 변경: Use of find_module() by the import system now raises ImportWarning.

버전 3.12에서 변경: find_module() has been removed. Use find_spec() instead.

5.4 로딩(loading)

모듈 스펙이 발견되면, 임포트 절차는 모듈을 로딩할 때 그것(그것이 가진 로더도)을 사용합니다. 여기에 임포트의 로딩 과정 동안 일어나는 일에 대한 대략적인 그림이 있습니다:

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # It is assumed 'exec_module' will also be defined on the loader.
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# The import-related module attributes get set here:
_init_module_attrs(spec, module)

if spec.loader is None:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # unsupported
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # namespace package
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

다음과 같은 세부 사항에 주의해야 합니다:

- 만약 주어진 이름의 모듈이 `sys.modules` 에 있다면, 임포트는 이미 그걸 돌려줄 겁니다.
- 로더가 모듈을 실행하기 전에 모듈은 `sys.modules` 에 자리를 잡습니다. 이것은 필수적인데 모듈이 (직접적 혹은 간접적으로) 자신을 임포트 할 수 있기 때문입니다; 먼저 `sys.modules` 에 추가함으로써 최악의 상황에 제한 없는 재귀(recursion)를 방지하고, 최선의 상황에 여러 번 로딩되는 것을 막습니다.
- 로딩이 실패하면, 실패한 모듈(오직 실패한 모듈만)은 `sys.modules` 에서 삭제됩니다. `sys.modules` 캐시에 이미 있는 모듈과 부수적 효과로 성공적으로 로딩된 모듈들은 캐시에 남아있어야만 합니다. 이는 실패한 모듈조차 `sys.modules` 에 남아있게 되는 리로딩과 대비됩니다.
- 모듈이 만들어졌지만, 아직 실행되기 전에, 뒤의 섹션 에서 요약되듯이, 임포트 절차는 임포트 관련 모듈 어트리뷰트들을 설정합니다(위의 의사 코드 예에서 "_init_module_attrs").
- 모듈 실행은 로딩에서 모듈의 이름 공간이 채워지는 결정적 순간입니다. 실행은 전적으로 로더에 위임되는데, 로더가 어떤 것이 어떻게 채워져야 하는지 결정합니다.
- 로딩 동안 만들어지고 exec_module() 로 전달되는 모듈은 임포트의 끝에 반환되는 것이 아닐 수 있습니다[2].

버전 3.4에서 변경: 임포트 시스템이 기초 공사에 대한 로더의 책임을 들고 갔습니다. 이것들은 전에는 `importlib.abc.Loader.load_module()` 메서드에서 수행되었습니다.

5.4.1 로더

모듈 로더는 로딩의 결정적인 기능을 제공합니다: 모듈 실행. 임포트 절차는 하나의 인자로 `importlib.abc.Loader.exec_module()` 메서드를 호출하는데, 실행할 모듈 객체가 전달됩니다. `exec_module()` 이 돌려주는 값은 무시됩니다.

로더는 다음과 같은 요구 조건들을 만족해야 합니다:

- 만약 모듈이 파이썬 모듈(내장 모듈이나 동적으로 로딩되는 확장이 아니라)이면, 로더는 모듈의 코드를 모듈의 전역 이름 공간(`module.__dict__`)에서 실행해야 합니다.
- 만약 로더가 모듈을 실행하지 못하면, `ImportError` 를 일으켜야 합니다. 하지만 `exec_module()` 동안 발생하는 다른 예외도 전파됩니다.

많은 경우에, 파인더와 로더는 같은 객체입니다; 그런 경우 `find_spec()` 메서드는 로더가 `self` 로 설정된 스펙을 돌려줍니다.

[2] importlib 구현은 반환 값을 직접 사용하지 않습니다. 대신에, `sys.modules` 에서 모듈 이름을 조회해서 모듈을 얻습니다. 이것의 간접적인 효과는 임포트되는 모듈이 `sys.modules` 에 있는 자신을 바꿀 수 있다는 것입니다. 이것은 구현 상세 동작이고 다른 파이썬 구현에서 동작한다고 보장되지 않습니다.

모듈 로더는 `create_module()` 메서드를 구현함으로써 로딩하는 동안 모듈 객체를 만드는 일에 개입할 수 있습니다. 하나의 인자, 모듈 스펙, 을 받아들이고 로딩 중 사용할 모듈 객체를 돌려줍니다. `create_module()` 은 모듈 객체의 어트리뷰트를 설정할 필요는 없습니다. 만약 메서드가 `None` 을 돌려주면, 임포트 절차는 새 모듈을 스스로 만듭니다.

Added in version 3.4: 로더의 `create_module()` 메서드.

버전 3.4에서 변경: `load_module()` 메서드는 `exec_module()` 로 대체되었고, 임포트 절차가 로딩의 공통 코드(boilerplate)에 대한 책임을 집니다.

이미 존재하는 로더들과의 호환을 위해, 임포트 절차는 `load_module()` 메서드가 존재하고, `exec_module()` 을 구현하지 않았으면 `load_module()` 을 사용합니다. 하지만 `load_module()` 은 폐지되었습니다. 로더는 대신 `exec_module()` 를 구현해야 합니다.

`load_module()` 메서드는 모듈을 실행하는 것 외에 위에서 언급한 모든 공통(boilerplate) 로딩 기능을 구현해야만 합니다. 같은 제약들이 모두 적용되는데, 추가적인 설명을 붙여보면:

- 만약 `sys.modules` 에 주어진 이름의 모듈 객체가 이미 존재하면, 로더는 반드시 그 객체를 사용해야 합니다. (그렇지 않으면, `importlib.reload()` 이 올바로 동작하지 않게 됩니다.) 만약 `sys.modules` 에 주어진 이름의 모듈이 없으면, 로더는 새 모듈객체를 만들고 `sys.modules` 에 추가해야 합니다.
- 제한 없는 재귀와 여러 번 로딩되는 것을 방지하기 위해, 로더가 모듈 코드를 실행하기 전에 모듈이 `sys.modules` 에 존재해야 합니다.
- 만약 로딩이 실패하면, 로더는 `sys.modules` 에 삽입한 모듈들을 제거해야 하는데, 실패한 모듈만을 제거해야 하고, 로더가 그 모듈을 직접 명시적으로 로드한 경우에만 그래야 합니다.

버전 3.5에서 변경: `exec_module()` 이 정의되었지만 `create_module()` 이 정의되지 않으면 `DeprecationWarning` 이 발생합니다.

버전 3.6에서 변경: `exec_module()` 이 정의되었지만 `create_module()` 이 정의되지 않으면 `ImportError` 를 일으킵니다.

버전 3.10에서 변경: Use of `load_module()` will raise `ImportWarning`.

5.4.2 서브 모듈

어떤 메커니즘으로든 (예를 들어, `importlib` API들, `import` 나 `import-from` 문, 내장 `__import__()`) 서브 모듈이 로드될 때, 서브 모듈 객체로의 연결은 부모 모듈의 이름 공간에 이루어집니다. 예를 들어, 패키지 `spam` 이 서브 모듈 `foo` 를 가지면, `spam.foo` 를 임포트 한 후에는 `spam` 이 서브 모듈에 연결된 어트리뷰트 `foo` 를 갖게 됩니다. 다음과 같은 디렉터리 구조로 되어 있다고 합시다:

```
spam/
    __init__.py
    foo.py
```

and `spam/__init__.py` has the following line in it:

```
from .foo import Foo
```

then executing the following puts name bindings for `foo` and `Foo` in the `spam` module:

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.Foo
<class 'spam.foo.Foo'>
```

파이썬의 익숙한 이름 연결 규칙에서 볼 때 의외의 결과로 보일 수 있습니다. 하지만 실제로는 임포트 시스템의 근본적인 기능입니다. 불변의 규칙은 이렇습니다: 만약 `sys.modules['spam']` 과 `sys.modules['spam.foo']` 가 있다면 (위의 임포트 이후의 상태가 그러합니다), 뒤에 있는 것은 반드시 앞에 있는 것의 `foo` 어트리뷰트가 되어야 합니다.

5.4.3 Module specs

임포트 절차는 임포트 동안 각 모듈에 대한 다양한 정보들을 사용합니다, 특히 로딩 전에. 대부분 정보는 모든 모듈의 공통이다. 모듈 스펙의 목적은 이 임포트 관련 정보를 모듈별로 요약하는 것입니다.

임포트 동안 스펙을 사용하면 상태가 임포트 시스템의 구성 요소들로 전달될 수 있습니다, 예를 들어 모듈 스펙을 만드는 파인더와 그것을 실행하는 로더 간에. 가장 중요한 것은, 임포트 절차가 로딩의 공통 연산 (boilerplate operation)을 수행할 수 있도록 하는 것입니다. 모듈 스펙이 없다면 로더가 모든 책임을 지게 됩니다.

The module's spec is exposed as *module.__spec__*. Setting __spec__ appropriately applies equally to *modules initialized during interpreter startup*. The one exception is __main__, where __spec__ is *set to None in some cases*.

See ModuleSpec for details on the contents of the module spec.

Added in version 3.4.

5.4.4 __path__ attributes on modules

The *__path__* attribute should be a (possibly empty) *sequence* of strings enumerating the locations where the package's submodules will be found. By definition, if a module has a __path__ attribute, it is a *package*.

A package's *__path__* attribute is used during imports of its subpackages. Within the import machinery, it functions much the same as sys.path, i.e. providing a list of locations to search for modules during import. However, __path__ is typically much more constrained than sys.path.

The same rules used for sys.path also apply to a package's __path__. sys.path_hooks (described below) are consulted when traversing a package's __path__.

A package's __init__.py file may set or alter the package's *__path__* attribute, and this was typically the way namespace packages were implemented prior to **PEP 420**. With the adoption of **PEP 420**, namespace packages no longer need to supply __init__.py files containing only __path__ manipulation code; the import machinery automatically sets __path__ correctly for the namespace package.

5.4.5 모듈 repr

기본적으로, 모든 모듈은 사용할만한 repr 을 갖고 있습니다. 하지만 위의 어트리뷰트들과 모듈 스펙에 있는 것들에 따라, 모듈 객체의 repr 을 좀 더 명시적으로 제어할 수 있습니다.

모듈이 스펙(__spec__)을 가지면, 임포트 절차는 그것으로부터 repr 을 만들려고 시도합니다. 그것이 실패하거나 스펙이 없으면, 임포트 시스템은 모듈에서 제공되는 것들로 기본 repr 을 구성합니다. module.__name__, module.__file__, module.__loader__ 을 repr 의 입력으로 사용하려고 시도하는데, 빠진 정보는 기본값으로 채웁니다.

사용되고 있는 정확한 규칙은 이렇습니다:

- 모듈이 __spec__ 어트리뷰트를 가지면, 스펙에 있는 정보로 repr 을 생성합니다. "name", "loader", "origin", "has_location" 어트리뷰트들이 사용됩니다.
- 모듈이 __file__ 어트리뷰트를 가지면, 모듈의 repr 의 일부로 사용됩니다.
- 모듈이 __file__ 어트리뷰트를 갖지 않지만 None 이 아닌 __loader__ 를 가지면, 로더의 repr 이 모듈의 repr 의 일부로 사용됩니다.
- 그렇지 않으면, repr 에 모듈의 __name__ 을 사용합니다.

버전 3.12에서 변경: Use of module_repr(), having been deprecated since Python 3.4, was removed in Python 3.12 and is no longer called during the resolution of a module's repr.

5.4.6 캐시된 바이트 코드 무효화

파이썬이 .pyc 파일로부터 캐시 된 바이트 코드를 로드하기 전에, 캐시가 최신 버전인지 소스 .py 파일과 비교합니다. 기본적으로, 파이썬은 소스의 최종 수정 타임스탬프와 크기를 캐시 파일을 만들 때 함께 저장해서 이 작업을 수행합니다. 실행시간에, 임포트 시스템은 캐시 파일에 저장된 메타 데이터를 소스의 메타 데이터와 대조하여 캐시 파일의 유효성을 검사합니다.

파이썬은 또한 "해시 기반" 캐시 파일을 지원하는데, 캐시 파일은 메타 데이터 대신에 소스 파일의 내용 해시를 저장합니다. 해시 기반 .pyc 파일에는 두 가지 변종이 있습니다: 검사형(checked)과 비검사형(unchecked). 검사형 해시 기반 .pyc 파일의 경우, 파이썬은 소스 파일을 해시하고 결과 해시를 캐시 파일의 해시와 비교하여 캐시 파일의 유효성을 검사합니다. 검사형 해시 기반 캐시 파일이 유효하지 않은 것으로 판명되면, 파이썬은 캐시 파일을 다시 생성하고 새로운 검사형 해시 기반 캐시 파일을 만듭니다. 비검사형 해시 기반 .pyc 파일의 경우, 파이썬은 단순히 캐시 파일이 존재할 경우 유효하다고 가정합니다. 해시 기반 .pyc 파일 유효성 검사 동작은 --check-hash-based-pycs 플래그로 재정의될 수 있습니다.

버전 3.7에서 변경: 해시 기반 .pyc 파일을 추가했습니다. 이전에는, 파이썬은 바이트 코드 캐시의 타임스탬프 기반 무효화만 지원했습니다.

5.5 경로 기반 파인더

앞에서 언급했듯이, 파이썬은 여러 기본 메타 경로 파인더들을 갖고 있습니다. 이 중 하나는, 경로 기반 파인더 (PathFinder) 로 불리는데, 경로 엔트리 들의 목록을 담고 있는 임포트 경로 를 검색합니다. 각 경로 엔트리는 모듈을 찾을 곳을 가리킵니다.

경로 기반 파인더 자신은 뭔가를 임포트하는 법에 대해서는 아는 것이 없습니다. 대신에, 각 경로 엔트리를 탐색하면서, 각각을 구체적인 경로 엔트리를 다루는 법을 아는 경로 엔트리 파인더와 관련시킵니다.

경로 엔트리 파인더의 기본 집합은 파일 시스템에서 모듈을 찾는데 필요한 모든 개념을 구현하는데, 파이썬 소스 코드(.py 파일들), 파이썬 바이트 코드(.pyc 파일들), 공유 라이브러리(예를 들어 .so 파일들)와 같은 특수 파일형들을 처리합니다. 표준라이브러리의 zipimport 모듈의 지원을 받으면, 기본 경로 엔트리 파인더는 이 모든 파일(공유 라이브러리를 제외한 것들)을 zip 파일들로부터 로딩합니다.

경로 엔트리가 파일 시스템의 위치로 제한될 필요는 없습니다. URL이나 데이터베이스 조회나 문자열로 지정될 수 있는 어떤 위치도 가능합니다.

경로 기반 파인더는 검색 가능한 경로 엔트리의 유형을 확장하고 커스터마이즈할 수 있도록 하는 추가의 훅과 프로토콜을 제공합니다. 예를 들어, 네트워크 URL을 경로 엔트리로 지원하고 싶다면, 웹에서 모듈을 찾는 HTTP 개념을 구현하는 훅을 작성할 수 있습니다. 이 훅 (콜러블)은 아래에서 설명하는 프로토콜을 지원하는 경로 엔트리 파인더 를 돌려주는데, 웹에 있는 모듈을 위한 로더를 얻는 데 사용됩니다.

경고의 글: 이 섹션과 앞에 나온 것들은 모두 파인더라는 용어를 사용하는데, 메타 경로 파인더 와 경로 엔트리 파인더 라는 용어를 사용해서 구분합니다. 이 두 종류의 파인더는 매우 유사해서 비슷한 프로토콜을 지원하고 임포트 절차에서 비슷한 방식으로 기능합니다. 하지만 이것들이 미묘하게 다르다는 것을 기억하는 것이 중요합니다. 특히, 메타 경로 파인더는 임포트 절차의 처음에 개입하는데, sys.meta_path 탐색을 통해 들어옵니다.

반면에, 경로 엔트리 파인더는 경로 기반 파인더의 구현 상세인데, 사실 경로 기반 파인더가 sys.meta_path 로 부터 제거되면, 경로 엔트리 파인더의 개념은 일절 호출되지 않습니다.

5.5.1 경로 엔트리 파인더

경로 기반 파인더 는 위치가 문자열 경로 엔트리 로 지정된 파이썬 모듈과 패키지를 찾고 로드하는 책임을 집니다. 대부분의 경로 엔트리는 파일 시스템의 위치를 가리키지만, 이것으로 한정될 필요는 없습니다.

메타 경로 파인더로서, 경로 기반 파인더 는 앞에서 설명한 find_spec() 프로토콜을 구현합니다. 하지만 모듈이 임포트 경로 에서 어떻게 발견되고 로드되는지는 커스터마이즈하는데 사용될 수 있는 추가의 훅을 제공합니다.

경로 기반 파인더 는 세 개의 변수를 사용합니다, sys.path, sys.path_hooks, sys.path_importer_cache. 패키지 객체의 __path__ 어트리뷰트 또한 사용된다. 이것들은 임포트 절차를 커스터마이즈할 수 있는 추가의 방법을 제공합니다.

sys.path contains a list of strings providing search locations for modules and packages. It is initialized from the PYTHONPATH environment variable and various other installation- and implementation-specific defaults. Entries in sys.path can name directories on the file system, zip files, and potentially other "locations" (see the site module) that should be searched for modules, such as URLs, or database queries. Only strings should be present on sys.path; all other data types are ignored.

경로 기반 파인더 는 메타 경로 파인더 이기 때문에, 앞에서 설명했듯이 임포트 절차는 경로 기반 파인더의 find_spec() 메서드를 호출하는 것으로 임포트 경로 검색을 시작합니다. find_spec() 에 제공되는

path 인자는 탐색할 문자열 경로들의 리스트입니다 - 보통 패키지 내에서 임포트 하면 패키지의 __path__ 어트리뷰트. path 인자가 None 이면, 최상위 임포트를 뜻하고 sys.path 가 사용됩니다.

The path based finder iterates over every entry in the search path, and for each of these, looks for an appropriate *path entry finder* (PathEntryFinder) for the path entry. Because this can be an expensive operation (e.g. there may be stat() call overheads for this search), the path based finder maintains a cache mapping path entries to path entry finders. This cache is maintained in sys.path_importer_cache (despite the name, this cache actually stores finder objects rather than being limited to *importer* objects). In this way, the expensive search for a particular *path entry* location' s *path entry finder* need only be done once. User code is free to remove cache entries from sys.path_importer_cache forcing the path based finder to perform the path entry search again.

경로 엔트리가 캐시에 없으면, 경로 기반 파인더는 sys.path_hooks 에 있는 모든 콜러블들을 탐색합니다. 이 목록의 각 경로 엔트리 훅 은 검색할 경로 엔트리 인자 한 개를 사용해서 호출됩니다. 이 콜러블은 경로 엔트리를 다룰 수 있는 경로 엔트리 파인더 를 돌려주거나, ImportError 를 발생시킬 수 있습니다. ImportError 는 경로 기반 파인더가 어떤 훅이 주어진 경로 엔트리 를 위한 경로 엔트리 파인더 를 발견할 수 없음을 알리는 데 사용합니다. 이 예외는 무시되고 임포트 경로 탐색은 계속됩니다. 훅은 문자열이나 바이트열을 기대해야 합니다; 바이트열의 인코딩은 훅이 결정하고(예를 들어, 파일 시스템 인코딩이나 UTF-8 이나 그 밖의 다른 것일 수 있습니다), 만약 훅이 인자를 디코딩할 수 없으면 ImportError 를 일으켜야 합니다.

만약 sys.path_hooks 탐색이 아무런 경로 엔트리 파인더 를 돌려주지 못하면, 경로 기반 파인더의 find_spec() 메서드는 sys.path_importer_cache 에 None 을 저장하고(이 경로 엔트리를 위한 파인더가 없음을 가리키기 위해), None 을 돌려줘서 이 메타 경로 파인더 가 모듈을 찾을 수 없음을 알립니다.

만약 sys.path_hooks 에 있는 어느 하나의 경로 엔트리 훅 콜러블이 경로 엔트리 파인더 를 돌려주면, 파인더에 모듈 스펙을 요청하기 위해 다음에 나오는 프로토콜이 사용됩니다. 모듈 스펙은 모듈을 로딩할 때 사용됩니다.

The current working directory – denoted by an empty string – is handled slightly differently from other entries on sys.path. First, if the current working directory cannot be determined or is found not to exist, no value is stored in sys.path_importer_cache. Second, the value for the current working directory is looked up fresh for each module lookup. Third, the path used for sys.path_importer_cache and returned by importlib.machinery.PathFinder.find_spec() will be the actual current working directory and not the empty string.

5.5.2 경로 엔트리 파인더 프로토콜

모듈과 초기화된 패키지의 임포트를 지원하고 이름 공간 패키지에 포션으로 이바지하기 위해, 경로 엔트리 파인더는 find_spec() 메서드를 구현해야 합니다.

find_spec() 은 두 개의 인자를 받아들입니다: 임포트 할 모듈의 완전히 정규화된 이름과 (생략 가능한) 타깃 모듈. find_spec() 은 값이 완전히 채워진 모듈의 스펙을 돌려줍니다. 이 스펙은 항상 "loader" 가 설정됩니다(한가지 예외가 있습니다).

To indicate to the import machinery that the spec represents a namespace *portion*, the path entry finder sets submodule_search_locations to a list containing the portion.

버전 3.4에서 변경: find_spec() replaced find_loader() and find_module(), both of which are now deprecated, but will be used if find_spec() is not defined.

예전의 경로 엔트리 파인더는 find_spec() 대신에 이 두 개의 폐지된 메서드들을 구현할 수 있습니다. 이 메서드들은 과거 호환성 때문에 아직도 사용됩니다. 하지만, find_spec() 이 경로 엔트리 파인더에 구현되면, 예전 메서드들은 무시됩니다.

find_loader() takes one argument, the fully qualified name of the module being imported. find_loader() returns a 2-tuple where the first item is the loader and the second item is a namespace *portion*.

임포트 프로토콜의 다른 구현들과의 과거 호환성을 위해, 많은 경로 엔트리 파인더들은 메타 경로 파인더가 지원하는 것과 같고 전통적인 find_module() 메서드 또한 지원합니다. 하지만 경로 엔트리 파인더 find_module() 메서드는 결코 path 인자로 호출되지 않습니다 (그것들은 경로 훅의 최초 호출 때 적절한 경로 정보를 기록해둘 것으로 기대됩니다).

경로 엔트리 파인더의 find_module() 메서드는 경로 엔트리 파인더가 이름 공간 패키지에 포션으로 이바지하는 것을 허락하지 않기 때문에 폐지되었습니다. 만약 경로 엔트리 파인더에 find_loader() 와

`find_module()` 이 모두 존재하면, 임포트 시스템은 항상 `find_module()` 대신 `find_loader()` 를 호출합니다.

버전 3.10에서 변경: Calls to `find_module()` and `find_loader()` by the import system will raise `ImportWarning`.

버전 3.12에서 변경: `find_module()` and `find_loader()` have been removed.

5.6 표준 임포트 시스템 교체하기

임포트 시스템 전체를 교체하기 위한 가장 신뢰성 있는 메커니즘은 `sys.meta_path` 의 기본값들을 모두 삭제하고, 새로 만든 메타 경로 훅들로 채우는 것입니다.

만약 임포트 시스템을 액세스하는 다른 API들에 영향을 주지 않고, 단지 임포트 문의 동작만을 변경해도 좋다면, 내장 `__import__()` 함수를 교체하는 것으로 충분할 수도 있습니다. 이 기법은 특정 모듈 내에서의 임포트 문의 동작만을 변경하도록 모듈 수준에서 적용될 수도 있습니다.

메타 경로의 앞쪽에 있는 훅에서 어떤 모듈의 임포트를 선택적으로 막으려면(표준 임포트 시스템을 완전히 비활성화하는 대신), `find_spec()` 에서 `None` 을 돌려주는 대신, `ModuleNotFoundError` 를 일으키는 것으로 충분합니다. 전자는 메타 경로 검색을 계속해야 한다는 것을 지시하는 반면, 예외를 일으키면 즉시 종료시킵니다.

5.7 패키지 상대 임포트

상대 임포트는 선행 점을 사용합니다. 단일 선행 점은 현재 패키지에서 시작하는 상대 임포트를 나타냅니다. 두 개 이상의 선행 점은 현재 패키지의 부모(들)에 대한 상대 임포트를 나타내며, 첫 번째 점 다음의 점 하나당 하나의 수준을 나타냅니다. 예를 들어, 다음과 같은 패키지 배치가 제공될 때:

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

`subpackage1/moduleX.py` 나 `subpackage1/__init__.py` 모두에서, 다음은 유효한 상대 임포트입니다:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

절대 임포트는 `import <>` 또는 `from <> import <>` 문법을 사용할 수 있지만, 상대 임포트는 두 번째 형식만 사용할 수 있습니다; 그 이유는:

```
import XXX.YYY.ZZZ
```

가 `XXX.YYY.ZZZ` 를 사용할 수 있는 표현식으로 노출하지만, .moduleY는 유효한 표현식이 아니기 때문입니다.

5.8 __main__ 에 대한 특별한 고려

__main__ 모듈은 파이썬의 임포트 시스템에서 특별한 경우입니다. 다른 곳에서 언급했듯이, __main__ 모듈은 sys 와 builtins 처럼 인터프리터 시작 때 직접 초기화됩니다. 하지만, 이 두 개와는 다르게, 이것은 엄밀하게 내장 모듈로 취급되지 않습니다. 이것은 __main__ 이 초기화되는 방식이 인터프리터를 실행할 때 주는 플래그와 다른 옵션들에 영향을 받기 때문입니다.

5.8.1 __main__.__spec__

__main__ 이 어떻게 초기화되는지에 따라, __main__.__spec__ 은 적절히 설정되기도 하고 None 으로 설정되기도 합니다.

파이썬이 -m 옵션으로 시작하면, __spec__ 은 해당하는 모듈이나 패키지의 모듈 스팩으로 설정됩니다. 또한 __spec__ 은 __main__ 모듈이 디렉터리나 zip 파일이나 다른 sys.path 엔트리를 실행하는 일부로 로드될 때 그 내용이 채워집니다.

나머지 경우 에는 __main__.__spec__ 은 None 으로 설정되는데, __main__ 을 채우는데 사용된 코드가 임포트 가능한 모듈에 직접 대응하지 않기 때문입니다:

- 대화형 프롬프트
- -c 옵션
- 표준 입력으로 실행
- 소스 파일이나 바이트 코드 파일로부터 직접 실행

마지막 경우에 __main__.__spec__ 이 항상 None 임에 주의해야 합니다. 설사 그 파일이 기술적으로 모듈로 임포트 될 수 있어도 그렇습니다. __main__ 에 올바른 모듈 메타데이터가 필요하다면 -m 스위치를 사용해야 합니다.

또한 __main__ 이 임포트 가능한 모듈에 대응되고, __main__.__spec__ 이 적절히 설정되었다 하더라도, 이 둘은 여전히 다른 모듈로 취급됨에 주의해야 합니다. 이것은 if __name__ == "__main__": 검사로 둘러싸인 블록이 모듈이 __main__ 이름 공간을 채울 때만 실행되고, 일반적인 임포트 때는 실행되지 않는다는 사실 때문입니다.

5.9 참고문헌

임포트 절차는 파이썬의 초창기부터 상당히 변해왔습니다. 문서를 작성한 이후에 약간의 세부사항이 변경되었기는 하지만, 최초의 패키지 규격 은 아직 읽을 수 있도록 남아있습니다.

sys.meta_path 의 최초 규격은 **PEP 302** 이고, 뒤이은 확장은 **PEP 420** 입니다.

PEP 420 introduced *namespace packages* for Python 3.3. **PEP 420** also introduced the find_loader() protocol as an alternative to find_module().

PEP 366 은 메인 모듈에서의 명시적인 상태 임포트를 위한 __package__ 어트리뷰트의 추가에 관해 설명하고 있습니다.

PEP 328 은 절대와 명시적인 상대 임포트들 도입하고 **PEP 366** 이 결국 __package__ 를 지정하게 되는 개념을 초기에 __name__ 으로 제안했습니다.

PEP 338 은 모듈을 스크립트로 실행하는 것을 정의합니다.

PEP 451 은 스팩 객체에 모듈별 임포트 상태를 요약하는 것을 추가합니다. 로더들에 주어졌던 대부분의 공통 코드 책임들을 임포트 절차로 옮기기도 했습니다. 이 변경은 임포트 시스템의 여러 API 들을 폐지하도록 만들었고, 파인더와 로더에 새 메서드들을 추가하기도 했습니다.

CHAPTER 6

표현식

이 장은 파이썬에서 사용되는 표현식 요소들의 의미를 설명합니다.

문법 유의 사항: 여기와 이어지는 장에서는, 어휘 분석이 아니라 문법을 설명하기 위해 확장 BNF 표기법을 사용합니다. 문법 규칙이 다음과 같은 형태를 가지고,

```
name: othername
```

뜻(semantics)을 주지 않으면, 이 형태의 `name` 의 뜻은 `othername` 과 같습니다.

6.1 산술 변환

When a description of an arithmetic operator below uses the phrase "the numeric arguments are converted to a common real type", this means that the operator implementation for built-in types works as follows:

- If both arguments are complex numbers, no conversion is performed;
- if either argument is a complex or a floating-point number, the other is converted to a floating-point number;
- 그렇지 않으면, 두 인자는 모두 정수여야 하고, 변환은 필요 없습니다.

어떤 연산자들(예를 들어, '%' 연산자의 왼쪽 인자로 주어지는 문자열)에 대해서는 몇 가지 추가의 규칙이 적용됩니다. 확장(extension)은 그들 자신의 변환 규칙을 정의해야 합니다.

6.2 아톰 (Atoms)

아톰은 표현식의 가장 기본적인 요소입니다. 가장 간단한 아톰은 식별자와 리터럴입니다. 괄호, 대괄호, 중괄호로 둘러싸인 형태도 문법적으로 아톰으로 분류됩니다. 아톰의 문법은 이렇습니다:

```
atom:      identifier | literal | enclosure
enclosure: parenth_form | list_display | dict_display | set_display
           | generator_expression | yield_atom
```

6.2.1 식별자 (이름)

아톰으로 등장하는 식별자는 이름입니다. 어휘 정의에 대해서는 식별자와 키워드 섹션을, 이름과 연결에 대한 문서는 이름과 연결(*binding*) 섹션을 보세요.

이름이 객체에 연결될 때, 아톰의 값을 구하면 객체가 나옵니다. 이름이 연결되지 않았을 때, 값을 구하려고 하면 `NameError` 예외가 일어납니다.

Private name mangling

When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class.

더 보기

The *class specifications*.

More precisely, private names are transformed to a longer form before code is generated for them. If the transformed name is longer than 255 characters, implementation-defined truncation may happen.

The transformation is independent of the syntactical context in which the identifier is used but only the following private identifiers are mangled:

- Any name used as the name of a variable that is assigned or read or any name of an attribute being accessed.

  The `__name__` attribute of nested functions, classes, and type aliases is however not mangled.

- The name of imported modules, e.g., `__spam` in `import __spam`. If the module is part of a package (i.e., its name contains a dot), the name is *not* mangled, e.g., the `__foo` in `import __foo.bar` is not mangled.

- The name of an imported member, e.g., `__f` in `from spam import __f`.

The transformation rule is defined as follows:

- The class name, with leading underscores removed and a single leading underscore inserted, is inserted in front of the identifier, e.g., the identifier `__spam` occurring in a class named `Foo`, `_Foo` or `__Foo` is transformed to `_Foo__spam`.

- If the class name consists only of underscores, the transformation is the identity, e.g., the identifier `__spam` occurring in a class named _ or __ is left as is.

6.2.2 리터럴 (Literals)

파이썬은 문자열과 바이트열 리터럴과 여러 가지 숫자 리터럴들을 지원합니다:

```
literal: stringliteral | bytesliteral
         | integer | floatnumber | imagnumber
```

Evaluation of a literal yields an object of the given type (string, bytes, integer, floating-point number, complex number) with the given value. The value may be approximated in the case of floating-point and imaginary (complex) literals. See section 리터럴 for details.

모든 리터럴은 불변 데이터형에 대응하기 때문에, 객체의 아이덴티티는 값 보다 덜 중요합니다. 같은 값의 리터럴에 대해 반복적으로 값을 구하면 (프로그램 텍스트의 같은 장소에 있거나 다른 장소에 있을 때) 같은 객체를 얻을 수도 있고, 같은 값의 다른 객체를 얻을 수도 있습니다.

6.2.3 괄호 안에 넣은 형

괄호 안에 넣은 형은, 괄호로 둘러싸인 생략 가능한 표현식 목록입니다:

```
parenth_form: "(" [starred_expression] ")"
```

괄호 안에 넣은 표현식 목록은, 무엇이건 그 표현식 목록이 산출하는 것이 됩니다: 목록이 적어도 하나의 쉼표를 포함하면, 튜플이 됩니다; 그렇지 않으면 표현식 목록을 구성한 단일 표현식이 됩니다.

빈 괄호 쌍은 빈 튜플 객체를 만듭니다. 튜플은 불변이기 때문에 리터럴에서와 같은 규칙이 적용됩니다 (즉, 두 개의 빈 튜플은 같은 객체일 수도 있고 그렇지 않을 수도 있습니다).

Note that tuples are not formed by the parentheses, but rather by use of the comma. The exception is the empty tuple, for which parentheses *are* required — allowing unparenthesized "nothing" in expressions would cause ambiguities and allow common typos to pass uncaught.

6.2.4 리스트, 집합, 딕셔너리의 디스플레이(display)

리스트, 집합, 딕셔너리를 구성하기 위해, 파이썬은 "디스플레이(displays)"라고 부르는 특별한 문법을 각기 두 가지 스타일로 제공합니다:

- 컨테이너의 내용을 명시적으로 나열하거나,
- 일련의 루프와 필터링 지시들을 통해 계산되는데, 컴프리헨션 (*comprehension*) 이라고 불립니다.

컴프리헨션의 공통 문법 요소들은 이렇습니다:

```
comprehension: assignment_expression comp_for
comp_for:      ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter:     comp_for | comp_if
comp_if:       "if" or_test [comp_iter]
```

컴프리헨션은 하나의 표현식과 그 뒤를 따르는 최소한 하나의 `for` 절과 없거나 여러 개의 `for` 또는 `if` 절로 구성됩니다. 이 경우, 새 컨테이너의 요소들은 각 `for` 또는 `if` 절이 왼쪽에서 오른쪽으로 중첩된 블록을 이루고, 가장 안쪽에 있는 블록에서 표현식의 값을 구해서 만들어낸 것들입니다.

하지만, 가장 왼쪽의 `for` 절에 있는 이터러블 표현식을 제외하고는, 컴프리헨션은 묵시적으로 중첩된 스코프에서 실행됩니다. 이렇게 해서 `target_list` 에서 대입되는 이름이 둘러싸는 스코프로 "누수" 되지 않도록 합니다.

가장 왼쪽의 `for` 절의 이터러블 표현식은, 둘러싸는 스코프에서 직접 평가된 다음, 묵시적으로 중첩된 스코프로 인자로 전달됩니다. 뒤따르는 `for` 절과 가장 왼쪽 `for` 절의 모든 필터 조건은, 가장 왼쪽 이터러블에서 얻은 값에 따라 달라질 수 있으므로 둘러싸는 스코프에서 평가할 수 없습니다. 예를 들면, `[x*y for x in range(10) for y in range(x, x+10)]`.

컴프리헨션이 항상 적절한 형의 컨테이너가 되게 하려고, 묵시적으로 중첩된 스코프에서 `yield` 와 `yield from` 표현식은 금지됩니다.

Since Python 3.6, in an *`async def`* function, an `async for` clause may be used to iterate over a *asynchronous iterator*. A comprehension in an `async def` function may consist of either a `for` or `async for` clause following the leading expression, may contain additional `for` or `async for` clauses, and may also use *`await`* expressions.

If a comprehension contains `async for` clauses, or if it contains `await` expressions or other asynchronous comprehensions anywhere except the iterable expression in the leftmost `for` clause, it is called an *asynchronous comprehension*. An asynchronous comprehension may suspend the execution of the coroutine function in which it appears. See also **PEP 530**.

Added in version 3.6: 비동기 컴프리헨션이 도입되었습니다.

버전 3.8에서 변경: `yield` 와 `yield from` 은 묵시적으로 중첩된 스코프에서 금지됩니다.

버전 3.11에서 변경: Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous.

6.2.5 리스트 디스플레이

리스트 디스플레이는 대괄호(square brackets)로 둘러싸인 표현식의 나열인데 비어있을 수 있습니다:

```
list_display: "[" [flexible_expression_list | comprehension] "]"
```

리스트 디스플레이는 리스트 객체를 만드는데, 그 내용은 표현식의 목록이나 컴프리헨션으로 지정할 수 있습니다. 쉼표로 분리된 표현식의 목록이 제공될 때, 그 요소들은 왼쪽에서 오른쪽으로 값이 구해지고, 그 순서대로 리스트 객체에 삽입됩니다. 컴프리헨션이 제공될 때, 리스트는 컴프리헨션으로 만들어지는 요소들로 구성됩니다.

6.2.6 집합 디스플레이

집합 디스플레이는 중괄호(curly braces)로 표시되고, 키와 값을 분리하는 콜론(colon)이 없는 것으로 딕셔너리 디스플레이와 구분될 수 있습니다.

```
set_display: "{" (flexible_expression_list | comprehension) "}"
```

집합 디스플레이는 새 가변 집합 객체를 만드는데, 그 내용은 표현식의 시퀀스나 컴프리헨션으로 지정됩니다. 쉼표로 분리된 표현식의 목록이 제공될 때, 그 요소들은 왼쪽에서 오른쪽으로 값이 구해지고, 집합 객체에 더해집니다. 컴프리헨션이 제공될 때, 집합은 컴프리헨션으로 만들어지는 요소들로 구성됩니다.

빈 집합은 `{}` 으로 만들어질 수 없습니다; 이 리터럴은 빈 딕셔너리를 만듭니다.

6.2.7 딕셔너리 디스플레이

A dictionary display is a possibly empty series of dict items (key/value pairs) enclosed in curly braces:

```
dict_display:       "{" [dict_item_list | dict_comprehension] "}"
dict_item_list:     dict_item ("," dict_item)* [","]
dict_item:          expression ":" expression | "**" or_expr
dict_comprehension: expression ":" expression comp_for
```

딕셔너리 디스플레이는 새 딕셔너리 객체를 만듭니다.

If a comma-separated sequence of dict items is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding value. This means that you can specify the same key multiple times in the dict item list, and the final dictionary' s value for that key will be the last one given.

A double asterisk `**` denotes *dictionary unpacking*. Its operand must be a *mapping*. Each mapping item is added to the new dictionary. Later values replace values already set by earlier dict items and earlier dictionary unpackings.

Added in version 3.5: **PEP 448** 에서 처음 제안된 딕셔너리 디스플레이로의 언 패킹.

딕셔너리 컴프리헨션은, 리스트와 집합 컴프리헨션에 대비해서, 일반적인 "for" 와 "if" 절 앞에 콜론으로 분리된 두 개의 표현식을 필요로 합니다. 컴프리헨션이 실행될 때, 만들어지는 키와 값 요소들이 만들어지는 순서대로 딕셔너리에 삽입됩니다.

Restrictions on the types of the key values are listed earlier in section 표준형 계층. (To summarize, the key type should be *hashable*, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last value (textually rightmost in the display) stored for a given key value prevails.

버전 3.8에서 변경: 파이썬 3.8 이전에는, 딕셔너리 컴프리헨션에서, 키와 값의 평가 순서가 잘 정의되어 있지 않았습니다. CPython에서, 값이 키보다 먼저 평가되었습니다. 3.8부터는, **PEP 572**의 제안에 따라 키가 값보다 먼저 평가됩니다.

6.2.8 제너레이터 표현식 (Generator expressions)

제너레이터 표현식은 괄호로 둘러싸인 간결한 제너레이터 표기법입니다.

```
generator_expression: "(" expression comp_for ")"
```

제너레이터 표현식은 새 제너레이터 객체를 만듭니다. 문법은 대괄호나 중괄호 대신 괄호로 둘러싸인다는 점만 제외하면 컴프리헨션과 같습니다.

제너레이터 표현식에 사용되는 변수들은 제너레이터 객체의 *__next__()* 메서드가 호출될 때 느긋하게 (lazily) 값이 구해집니다 (일반 제너레이터와 마찬가지입니다). 그러나 가장 왼쪽의 `for` 절에 있는 이터러블 표현식은 즉시 값이 구해져서, 그것으로 인해 발생하는 에러는 첫 번째 값이 검색되는 지점이 아니라 제너레이터 표현식이 정의된 지점에서 발생합니다. 후속 `for` 절과 가장 왼쪽 `for` 절의 모든 필터 조건은, 가장 왼쪽 이터러블에서 가져온 값에 따라 달라질 수 있으므로 둘러싸는 스코프에서 평가할 수 없습니다. 예를 들어: `(x*y for x in range(10) for y in range(x, x+10))`.

단지 하나의 인자만 갖는 호출에서는 괄호를 생략할 수 있습니다. 자세한 내용은 호출 섹션을 보세요.

제너레이터 표현식 자체의 기대되는 연산을 방해하지 않기 위해, 묵시적으로 정의된 제너레이터에서 `yield` 와 `yield from` 표현식은 금지됩니다.

제너레이터 표현식이 `async for` 절이나 *await* 표현식을 포함하면 비동기 제너레이터 표현식 (*asynchronous generator expression*) 이라고 불립니다. 비동기 제너레이터 표현식은 새 비동기 제너레이터 객체를 돌려주는데 이것은 비동기 이터레이터입니다 (비동기 이터레이터 (*Asynchronous Iterators*) 를 참조하세요).

Added in version 3.6: 비동기식 제너레이터 표현식이 도입되었습니다.

버전 3.7에서 변경: 파이썬 3.7 이전에는, 비동기 제너레이터 표현식이 *async def* 코루틴에만 나타날 수 있었습니다. 3.7부터는, 모든 함수가 비동기식 제너레이터 표현식을 사용할 수 있습니다.

버전 3.8에서 변경: `yield` 와 `yield from` 은 묵시적으로 중첩된 스코프에서 금지됩니다.

6.2.9 일드 표현식(Yield expressions)

```
yield_atom:       "(" yield_expression ")"
yield_from:       "yield" "from" expression
yield_expression: "yield" yield_list | yield_from
```

The yield expression is used when defining a *generator* function or an *asynchronous generator* function and thus can only be used in the body of a function definition. Using a yield expression in a function' s body causes that function to be a generator function, and using it in an *async def* function' s body causes that coroutine function to be an asynchronous generator function. For example:

```
def gen():  # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    yield 123
```

둘러싸는 스코프에 대한 부작용으로 인해, `yield` 표현식은 컴프리헨션과 제너레이터 표현식을 구현하는 데 사용되는 묵시적으로 정의된 스코프에 사용될 수 없습니다.

버전 3.8에서 변경: 일드 표현식은 컴프리헨션과 제너레이터 표현식을 구현하는 데 사용되는 묵시적으로 정의된 스코프에서 금지됩니다.

제너레이터 함수는 다음에서 설명합니다. 반면에 비동기 제너레이터 함수는 비동기 제너레이터 함수 섹션에서 별도로 설명합니다.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of the generator function. The execution starts when one of the generator' s methods is called. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of *yield_list* to the generator' s caller, or `None` if *yield_list* is omitted. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator' s methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If *__next__()* is used (typically via either a *for* or the `next()` builtin) then the result is `None`. Otherwise, if *send()* is used, then the result will be the value passed in to that method.

이 모든 것들은 제너레이터 함수를 코루틴과 아주 비슷하게 만듭니다; 여러 번 결과를 만들고, 하나 이상의 진입 지점을 갖고 있으며, 실행이 일시 중지될 수 있습니다. 유일한 차이점은 제너레이터 함수는 yield 한 후에 실행이 어디에서 계속되어야 하는지를 제어할 수 없다는 점입니다; 제어는 항상 제너레이터의 호출자로 전달됩니다.

일드 표현식은 *try* 구조물의 어디에서건 허락됩니다. 제너레이터가 (참조 횟수가 0이 되거나 가비지 수거됨으로써) 파이널라이즈(finalize)되기 전에 재개되지 않으면, 제너레이터-이터레이터의 *close()* 메서드가 호출되어, 대기 중인 *finally* 절이 실행되도록 허락합니다.

`yield from <expr>` 이 사용될 때, 제공된 표현식은 어터러블이어야 합니다. 그 이터러블을 이터레이트해서 생성되는 값들은 현재 제너레이터 메서드의 호출자에게 바로 전달됩니다. *send()* 로 전달된 모든 값과 *throw()* 로 전달된 모든 예외는 밑에 있는(underlying) 이터레이터가 해당 메서드를 갖고 있다면 그곳으로 전달됩니다. 그렇지 않다면, *send()* 는 `AttributeError` 나 `TypeError` 를 일으키지만, *throw()* 는 전달된 예외를 즉시 일으킨다.

밑에 있는 이러레이터가 완료될 때, 발생하는 `StopIteration` 인스턴스의 `value` 어트리뷰트는 일드 표현식의 값이 됩니다. `StopIteration` 를 일으킬 때 명시적으로 설정되거나, 서브 이터레이터가 제너레이터일 경우는 자동으로 이루어집니다 (서브 제너레이터가 값을 돌려(return)줌으로써).

버전 3.3에서 변경: 서브 이터레이터로 제어 흐름을 위임하는 `yield from <expr>` 를 추가했습니다.

일드 표현식이 대입문의 우변에 홀로 나온다면 괄호를 생략할 수 있습니다.

더 보기

PEP 255 - 간단한 제너레이터
파이썬에 제너레이터와 *yield* 문을 추가하는 제안.

PEP 342 - 개선된 제너레이터를 통한 코루틴
제너레이터의 API와 문법을 개선해서, 간단한 코루틴으로 사용할 수 있도록 만드는 제안.

PEP 380 - 서브 제너레이터로 위임하는 문법
The proposal to introduce the *yield_from* syntax, making delegation to subgenerators easy.

PEP 525 - 비동기 제너레이터
코루틴 함수에 제너레이터 기능을 추가하여 **PEP 492**을 확장한 제안.

제너레이터-이터레이터 메서드

이 서브섹션은 제너레이터 이터레이터의 메서드들을 설명합니다. 제너레이터 함수의 실행을 제어하는데 사용될 수 있습니다.

제너레이터가 이미 실행 중일 때 아래에 나오는 메서드들을 호출하면 `ValueError` 예외를 일으키는 것에 주의해야 합니다.

`generator.__next__()`

Starts the execution of a generator function or resumes it at the last executed yield expression. When a generator function is resumed with a *__next__()* method, the current yield expression always evaluates to `None`. The execution then continues to the next yield expression, where the generator is suspended again, and the value of the *yield_list* is returned to *__next__()*' s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

이 메서드는 보통 묵시적으로 호출됩니다, 예를 들어, *for* 루프나 내장 `next()` 함수에 의해.

`generator.send(`*value*`)`

실행을 재개하고 제너레이터 함수로 값을 "보냅니다(send)". *value* 인자는 현재 일드 표현식의 값이 됩니다. *send()* 메서드는 제너레이터가 yield 하는 다음 값을 돌려주거나, 제너레이터가 다른 값을 yield 하지 않고 종료하면 `StopIteration` 을 일으킵니다. *send()* 가 제너레이터를 시작시키도록 호출될 때, 값을 받을 일드 표현식이 없으므로, 인자로는 반드시 `None` 을 전달해야 합니다.

`generator.throw(`*value*`)`

`generator.throw(`*type*`[,` *value*`[,` *traceback*`]])`

Raises an exception at the point where the generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

In typical use, this is called with a single exception instance similar to the way the *raise* keyword is used.

For backwards compatibility, however, the second signature is supported, following a convention from older versions of Python. The *type* argument should be an exception class, and *value* should be an exception instance. If the *value* is not provided, the *type* constructor is called to get an instance. If *traceback* is provided, it is set on the exception, otherwise any existing `__traceback__` attribute stored in *value* may be cleared.

버전 3.12에서 변경: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

`generator.close()`

Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function catches the exception and returns a value, this value is returned from *close()*. If the generator function is already closed, or raises `GeneratorExit` (by not catching the exception), *close()* returns `None`. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. If the generator has already exited due to an exception or normal exit, *close()* returns `None` and has no other effect.

버전 3.13에서 변경: If a generator returns a value upon being closed, the value is returned by *close()*.

사용 예

여기에 제너레이터와 제너레이터 함수의 동작을 시연하는 간단한 예가 있습니다:

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
...         print("Don't forget to clean up when 'close()' is called.")
...
>>> generator = echo(1)
>>> print(next(generator))
Execution starts when 'next()' is called for the first time.
1
>>> print(next(generator))
None
>>> print(generator.send(2))
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

`yield from` 을 사용하는 예는, “What’ s New in Python.” 에 있는 pep-380 을 보세요.

비동기 제너레이터 함수

async def 를 사용한 함수나 메서드에서 일드 표현식의 존재는 그 함수를 비동기 제너레이터 함수로 정의합니다.

비동기 제너레이터 함수가 호출되면, 비동기 제너레이터 객체로 알려진 비동기 이터레이터를 돌려줍니다. 그런 다음 그 객체는 제너레이터 함수의 실행을 제어합니다. 비동기 제너레이터 객체는 보통 코루틴 함수의 *async for* 문에서 사용되는데, 제너레이터 객체가 *for* 문에서 사용되는 방식과 유사합니다.

Calling one of the asynchronous generator’ s methods returns an *awaitable* object, and the execution starts when this object is awaited on. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of *yield_list* to the awaiting coroutine. As with a generator, suspension means that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by awaiting on the next object returned by the asynchronous generator’ s methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If *__anext__()* is used then the result is `None`. Otherwise, if *asend()* is used, then the result will be the value passed in to that method.

If an asynchronous generator happens to exit early by *break*, the caller task being cancelled, or other exceptions, the generator’ s async cleanup code will run and possibly raise exceptions or access context variables in an unexpected context–perhaps after the lifetime of tasks it depends, or during the event loop shutdown when the async-generator garbage collection hook is called. To prevent this, the caller must explicitly close the async generator by calling *aclose()* method to finalize the generator and ultimately detach it from the event loop.

비동기 제너레이터 함수에서, 일드 표현식은 *try* 구조물의 어디에서건 허락됩니다. 하지만, 비동기 제너레이터가 (참조 횟수가 0이 되거나 가비지 수거됨으로써) 파이널라이즈(finalize)되기 전에 재개되지 않으면, `try` 구조물 내의 일드 표현식은 대기 중인 *finally* 절을 실행하는 데 실패할 수 있습니다. 이 경우에, 비동기 제너레이터-이터레이터의 *aclose()* 를 호출하고, 그 결과로 오는 코루틴 객체를 실행해서, 대기

중인 `finally` 절이 실행되도록 하는 책임은, 비동기 제너레이터를 실행하는 이벤트 루프(event loop) 나 스케줄러(scheduler)에게 있습니다.

To take care of finalization upon event loop termination, an event loop should define a *finalizer* function which takes an asynchronous generator-iterator and presumably calls *aclose()* and executes the coroutine. This *finalizer* may be registered by calling `sys.set_asyncgen_hooks()`. When first iterated over, an asynchronous generator-iterator will store the registered *finalizer* to be called upon finalization. For a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py.

표현식 `yield from <expr>` 를 비동기 제너레이터 함수에서 사용하는 것은 문법 에러다.

비동기 제너레이터-이터레이터 메서드

이 서브섹션은 비동기 제너레이터 이터레이터의 메서드를 설명하는데, 제너레이터 함수의 실행을 제어하는 데 사용됩니다.

async agen.**__anext__**()

Returns an awaitable which when run starts to execute the asynchronous generator or resumes it at the last executed yield expression. When an asynchronous generator function is resumed with an *__anext__()* method, the current yield expression always evaluates to `None` in the returned awaitable, which when run will continue to the next yield expression. The value of the *yield_list* of the yield expression is the value of the `StopIteration` exception raised by the completing coroutine. If the asynchronous generator exits without yielding another value, the awaitable instead raises a `StopAsyncIteration` exception, signalling that the asynchronous iteration has completed.

이 메서드는 보통 *async for* 루프에 의해 묵시적으로 호출됩니다.

async agen.**asend**(*value*)

Returns an awaitable which when run resumes the execution of the asynchronous generator. As with the *send()* method for a generator, this "sends" a value into the asynchronous generator function, and the *value* argument becomes the result of the current yield expression. The awaitable returned by the *asend()* method will return the next value yielded by the generator as the value of the raised `StopIteration`, or raises `StopAsyncIteration` if the asynchronous generator exits without yielding another value. When *asend()* is called to start the asynchronous generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

async agen.**athrow**(*value*)

async agen.**athrow**(*type*[, *value*[, *traceback*]])

어웨이터블을 돌려주는데, 비동기 제너레이터가 일시 중지한 지점에 `type` 형의 예외를 일으키고, 제너레이터 함수가 yield 한 다음 값을 발생하는 `StopIteration` 예외의 값으로 돌려줍니다. 비동기 제너레이터가 다른 값을 yield 하지 않고 종료하면, 어웨이터블에 의해 `StopAsyncIteration` 예외가 일어납니다. 제너레이터 함수가 전달된 예외를 잡지 않거나, 다른 예외를 일으키면, 어웨이터블을 실행할 때 그 예외가 어웨이터블의 호출자에게 퍼집니다.

버전 3.12에서 변경: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

async agen.**aclose**()

어웨이터블을 돌려주는데, 실행하면, 비동기 제너레이터 함수가 일시 정지한 지점으로 `GeneratorExit` 를 던집니다. 만약 그 이후에 비동기 제너레이터 함수가 우아하게 (gracefully) 종료하거나, 이미 닫혔거나, (그 예외를 잡지 않음으로써) `GeneratorExit` 를 일으키면, 돌려준 어웨이터블은 `StopIteration` 예외를 일으킵니다. 이어지는 비동기 제너레이터 호출이 돌려주는 추가의 어웨이터블들은 `StopAsyncIteration` 예외를 일으킵니다. 만약 비동기 제너레이터가 값을 yield 하면 어웨이터블에 의해 `RuntimeError` 가 발생합니다. 만약 비동기 제너레이터가 그 밖의 다른 예외를 일으키면, 어웨이터블의 호출자로 퍼집니다. 만약 비동기 제너레이터가 예외나 정상 종료로 이미 종료했으면, 더 이어지는 *aclose()* 호출은 아무것도 하지 않는 어웨이터블을 돌려줍니다.

6.3 프라이머리

프라이머리는 언어에서 가장 강하게 결합하는 연산들을 나타냅니다. 문법은 이렇습니다:

```
primary: atom | attributeref | subscription | slicing | call
```

6.3.1 어트리뷰트 참조

어트리뷰트 참조는 마침표(period)와 이름이 뒤에 붙은 프라이머리다:

```
attributeref: primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, which most objects do. This object is then asked to produce the attribute whose name is the identifier. The type and value produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

This production can be customized by overriding the *__getattribute__()* method or the *__getattr__()* method. The `__getattribute__()` method is called first and either returns a value or raises `AttributeError` if the attribute is not available.

If an `AttributeError` is raised and the object has a `__getattr__()` method, that method is called as a fallback.

6.3.2 서브스크립션(Subscriptions)

The subscription of an instance of a *container class* will generally select an element from the container. The subscription of a *generic class* will generally return a GenericAlias object.

```
subscription: primary "[" flexible_expression_list "]"
```

When an object is subscripted, the interpreter will evaluate the primary and the expression list.

The primary must evaluate to an object that supports subscription. An object may support subscription through defining one or both of *__getitem__()* and *__class_getitem__()*. When the primary is subscripted, the evaluated result of the expression list will be passed to one of these methods. For more details on when `__class_getitem__` is called instead of `__getitem__`, see *__class_getitem__ versus __getitem__*.

If the expression list contains at least one comma, or if any of the expressions are starred, the expression list will evaluate to a `tuple` containing the items of the expression list. Otherwise, the expression list will evaluate to the value of the list' s sole member.

버전 3.11에서 변경: Expressions in an expression list may be starred. See **PEP 646**.

For built-in objects, there are two types of objects that support subscription via *__getitem__()*:

1. Mappings. If the primary is a *mapping*, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. An example of a builtin mapping class is the `dict` class.
2. Sequences. If the primary is a *sequence*, the expression list must evaluate to an `int` or a `slice` (as discussed in the following section). Examples of builtin sequence classes include the `str`, `list` and `tuple` classes.

The formal syntax makes no special provision for negative indices in *sequences*. However, built-in sequences all provide a *__getitem__()* method that interprets negative indices by adding the length of the sequence to the index so that, for example, `x[-1]` selects the last item of `x`. The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero). Since the support for negative indices and slicing occurs in the object' s *__getitem__()* method, subclasses overriding this method will need to explicitly add that support.

A `string` is a special kind of sequence whose items are *characters*. A character is not a separate data type but a string of exactly one character.

6.3.3 슬라이싱(Slicings)

슬라이싱은 시퀀스 객체 (예를 들어, 문자열 튜플 리스트)에서 어떤 범위의 항목들을 선택합니다. 슬라이싱은 표현식이나 대입의 타깃이나 *del* 문에 사용될 수 있습니다. 슬라이싱의 문법은 이렇습니다:

```
slicing:      primary "[" slice_list "]"
slice_list:   slice_item ("," slice_item)* [","]
slice_item:   expression | proper_slice
proper_slice: [lower_bound] ":" [upper_bound] [ ":" [stride] ]
lower_bound:  expression
upper_bound:  expression
stride:       expression
```

이 형식 문법에는 모호함이 있습니다: 표현식 목록처럼 보이는 것들은 모두 슬라이스 목록으로 보이기도 해서, 모든 서브스크립션이 슬라이싱으로 해석될 수도 있습니다. 문법을 더 복잡하게 만드는 대신, 이 경우에 서브스크립션으로 해석하는 것이 슬라이싱으로 해석하는 것에 우선한다고 정의하는 것으로 애매함을 제거합니다 (이 경우는 슬라이스 목록이 고유한 슬라이스(proper slice) 를 하나도 포함하지 않을 때입니다).

The semantics for a slicing are as follows. The primary is indexed (using the same *__getitem__()* method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section 표준형 계층) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

6.3.4 호출

호출은 콜러블 객체 (예를 들어, 함수) 를 빌 수도 있는 인자 들의 목록으로 호출합니다.

```
call:                 primary "(" [argument_list [","] | comprehension] ")"
argument_list:        positional_arguments ["," starred_and_keywords]
                        ["," keywords_arguments]
                      | starred_and_keywords ["," keywords_arguments]
                      | keywords_arguments
positional_arguments: positional_item ("," positional_item)*
positional_item:      assignment_expression | "*" expression
starred_and_keywords: ("*" expression | keyword_item)
                      ("," "*" expression | "," keyword_item)*
keywords_arguments:   (keyword_item | "**" expression)
                      ("," keyword_item | "," "**" expression)*
keyword_item:         identifier "=" expression
```

생략할 수 있는 마지막 쉼표가 위치나 키워드 인자 뒤에 나타날 수 있지만, 의미를 바꾸지 않습니다.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and all objects having a *__call__()* method are callable). All argument expressions are evaluated before the call is attempted. Please refer to section 함수 정의 for the syntax of formal *parameter* lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a `TypeError` exception is raised. Otherwise, the argument is placed in the slot, filling it (even if the expression is `None`, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don' t specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

구현은 위치 매개변수가 이름을 갖지 않아서, 설사 문서화의 목적으로 이름이 붙여졌다 하더라도, 키워드로 공급될 수 없는 내장 함수들을 제공할 수 있습니다. CPython 에서, 인자들을 파싱하기 위해 `PyArg_ParseTuple()` 를 사용하는 C로 구현된 함수들이 이 경우입니다.

형식 매개변수 슬롯들보다 많은 위치 인자들이 있으면, `*identifier` 문법을 사용하는 형식 매개변수가 있지 않은 한, `TypeError` 예외를 일으킵니다; 이 경우, 그 형식 매개변수는 남는 위치 인자들을 포함하는 튜플을 전달받습니다 (또는 남는 위치 인자들이 없으면 빈 튜플).

키워드 인자가 형식 매개변수 이름에 대응하지 않으면, `**identifier` 문법을 사용하는 형식 매개변수가 있지 않은 한, `TypeError` 예외를 일으킵니다; 이 경우, 그 형식 매개변수는 남는 키워드 인자들을 포함하는 딕셔너리나, 남는 위치기반 인자들이 없으면 빈 (새) 딕셔너리를 전달받습니다.

문법 `*expression` 이 함수 호출에 등장하면, `expression` 의 값은 이터러블 이 되어야 합니다. 이 이터러블의 요소들은, 그것들이 추가의 위치 인자들인 것처럼 취급됩니다. 호출 `f(x1, x2, *y, x3, x4)` 의 경우, *y* 의 값을 구할 때 시퀀스 *y1*, …, *yM* 이 나온다면, 이것은 M+4개의 위치 인자들 *x1*, *x2*, *y1*, …, *yM*, *x3*, *x4* 로 호출하는 것과 동등합니다.

이로 인한 결과는 설사 `*expression` 문법이 명시적인 키워드 인자 뒤에 나올 수는 있어도, 키워드 인자 (그리고 모든 `**expression` 인자들 – 아래를 보라) 전에 처리된다는 것입니다. 그래서:

```
>>> def f(a, b):
...     print(a, b)
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the `*expression` syntax to be used in the same call, so in practice this confusion does not often arise.

If the syntax `**expression` appears in the function call, `expression` must evaluate to a *mapping*, the contents of which are treated as additional keyword arguments. If a parameter matching a key has already been given a value (by an explicit keyword argument, or from another unpacking), a `TypeError` exception is raised.

When `**expression` is used, each key in this mapping must be a string. Each value from the mapping is assigned to the first formal parameter eligible for keyword assignment whose name is equal to the key. A key need not be a Python identifier (e.g. `"max-temp °F"` is acceptable, although it will not match any formal parameter that could be declared). If there is no match to a formal parameter the key-value pair is collected by the `**` parameter, if there is one, or if there is not, a `TypeError` exception is raised.

문법 `*identifier` 이나 `**identifier` 를 사용하는 형식 매개변수들은 위치 인자 슬롯이나 키워드 인자 아름들로 사용될 수 없습니다.

버전 3.5에서 변경: 함수 호출은 임의의 개수의 `*` and `**` 언 패킹을 받아들이고, 위치 인자들이 이터러블 언 패킹 (`*`) 뒤에 올 수 있고, 키워드 인자가 딕셔너리 언 패킹 (`**`) 뒤에 올 수 있습니다. 최초로 **PEP 448** 에서 제안되었습니다.

호출은 예외를 일으키지 않는 한, 항상 어떤 값을 돌려줍니다, `None` 일 수 있습니다. 이 값이 어떻게 계산되는지는 콜러블 객체의 형에 달려있습니다.

만약 그것이—

사용자 정의 함수면:

The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section 함수 정의. When the code block executes a *return* statement, this specifies the return value of the function call. If execution reaches the end of the code block without executing a *return* statement, the return value is `None`.

내장 함수나 메서드면:
결과는 인터프리터에 달려있습니다; 내장 함수와 메서드들에 대한 설명은 built-in-funcs 를 보세요.

클래스 객체면:
그 클래스의 새 인스턴스가 반환됩니다.

클래스 인스턴스 메서드면:
대응하는 사용자 정의 함수가 호출되는데, 그 인스턴스가 첫 번째 인자가 되는 하나만큼 더 긴 인자 목록이 전달됩니다.

클래스 인스턴스면:
The class must define a *__call__()* method; the effect is then the same as if that method was called.

6.4 어웨이트 표현식

어웨이터블 에서 코루틴 의 실행을 일시 중지합니다. 오직 코루틴 함수 에서만 사용할 수 있습니다.

```
await_expr: "await" primary
```

Added in version 3.5.

6.5 거듭제곱 연산자

거듭제곱 연산자는 그것의 왼쪽에 붙는 일 항 연산자보다 더 강하게 결합합니다; 그것의 오른쪽에 붙는 일 항 연산자보다는 약하게 결합합니다. 문법은 이렇습니다:

```
power: (await_expr | primary) ["**" u_expr]
```

그래서, 괄호가 없는 거듭제곱과 일 항 연산자의 시퀀스에서, 연산자는 오른쪽에서 왼쪽으로 값이 구해집니다 (이것이 피연산자의 값을 구하는 순서를 제약하는 것은 아닙니다): `-1**2` 은 `-1` 이 됩니다.

거듭제곱 연산자는 내장 `pow()` 함수가 두 개의 인자로 호출될 때와 같은 의미가 있습니다: 왼쪽 인자를 오른쪽 인자만큼 거듭제곱한 값을 줍니다. 숫자 인자는 먼저 공통 형으로 변환되고, 결과는 그 형입니다.

int 피연산자의 경우, 두 번째 인자가 음수가 아닌 이상 결과는 피연산자들과 같은 형을 갖습니다; 두 번째 인자가 음수면, 모든 인자는 float로 변환되고, float 결과가 전달됩니다. 예를 들어, `10**2` 는 `100` 를 돌려주지만, `10**-2` 는 `0.01` 를 돌려줍니다.

`0.0` 를 음수로 거듭제곱하면 `ZeroDivisionError` 를 일으킵니다. 음수를 분수로 거듭제곱하면 복소수 (`complex`) 가 나옵니다. (예전 버전에서는 `ValueError` 를 일으켰습니다.)

This operation can be customized using the special *__pow__()* and *__rpow__()* methods.

6.6 일 항 산술과 비트 연산

모든 일 항 산술과 비트 연산자는 같은 우선순위를 갖습니다.

```
u_expr: power | "-" u_expr | "+" u_expr | "~" u_expr
```

The unary `-` (minus) operator yields the negation of its numeric argument; the operation can be overridden with the *__neg__()* special method.

The unary `+` (plus) operator yields its numeric argument unchanged; the operation can be overridden with the *__pos__()* special method.

The unary `~` (invert) operator yields the bitwise inversion of its integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers or to custom objects that override the *__invert__()* special method.

세 가지 경우 모두, 인자가 올바른 형을 갖지 않는다면, `TypeError` 예외가 발생합니다.

6.7 이항 산술 연산

이항 산술 연산자는 관습적인 우선순위를 갖습니다. 이 연산자 중 일부는 일부 비 숫자 형에도 적용됨에 주의해야 합니다. 거듭제곱 연산자와는 별개로, 오직 두 가지 수준만 있는데, 하나는 곱셈형 연산자들이고, 하나는 덧셈형 연산자들입니다.

```
m_expr: u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
        m_expr "//" u_expr | m_expr "/" u_expr |
        m_expr "%" u_expr
a_expr: m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer and the other must be a sequence. In the former case, the numbers are converted to a common real type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

This operation can be customized using the special *__mul__()* and *__rmul__()* methods.

버전 3.14에서 변경: If only one operand is a complex number, the other operand is converted to a floating-point number.

`@` (at) 연산자는 행렬 곱셈에 사용하려는 것입니다. 파이썬의 내장형들 어느 것도 이 연산자를 구현하지 않습니다.

This operation can be customized using the special *__matmul__()* and *__rmatmul__()* methods.

Added in version 3.5.

`/` (나눗셈)과 `//` (정수 나눗셈, floor division) 연산자들은 그 인자들의 몫(quotient)을 줍니다. 숫자 인자들은 먼저 공통형으로 변환됩니다. 정수들의 나눗셈은 실수를 만드는 반면, 정수들의 정수 나눗셈은 정숫값을 줍니다; 그 결과는 수학적인 나눗셈의 결과에 'floor' 함수를 적용한 것입니다. 0으로 나누는 것은 `ZeroDivisionError` 예외를 일으킵니다.

The division operation can be customized using the special *__truediv__()* and *__rtruediv__()* methods. The floor division operation can be customized using the special *__floordiv__()* and *__rfloordiv__()* methods.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating-point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand[1].

정수 나눗셈과 모듈로 연산자는 다음과 같은 항등식으로 연결되어 있습니다: `x == (x//y)*y + (x%y)`. 정수 나눗셈과 모듈로는 내장 함수 `divmod()` 와도 연결되어 있습니다: `divmod(x, y) == (x//y, x%y)`.[2].

숫자들에 대해 모듈로 연산을 수행하는 것에 더해, `%` 연산자는 예전 스타일의 문자열 포매팅 (인터폴레이션이라고도 알려져 있습니다)을 수행하기 위해 문자열 객체에 의해 다시 정의됩니다. 문자열 포매팅의 문법은 파이썬 라이브러리 레퍼런스의 섹션 old-string-formatting 에서 설명합니다.

The *modulo* operation can be customized using the special *__mod__()* and *__rmod__()* methods.

The floor division operator, the modulo operator, and the `divmod()` function are not defined for complex numbers. Instead, convert to a floating-point number using the `abs()` function if appropriate.

The `+` (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both be sequences of the same type. In the former case, the numbers are converted to a common real type and then added together. In the latter case, the sequences are concatenated.

This operation can be customized using the special *__add__()* and *__radd__()* methods.

[1] `abs(x%y) < abs(y)` 이 수학적으로는 참이지만, float의 경우에는 소수점 자름(roundoff) 때문에 수치적으로 참이 아닐 수 있습니다. 예를 들어, 파이썬 float가 IEEE 754 배정도 숫자인 플랫폼을 가정할 때, `-1e-100 % 1e100` 가 `1e100` 와 같은 부호를 가지기 위해, 계산된 결과는 `-1e-100 + 1e100` 인데, 수치적으로는 `1e100` 과 정확히 같은 값입니다. 함수 `math.fmod()` 는 부호가 첫 번째 인자의 부호에 맞춰진 결과를 주기 때문에, 이 경우 `-1e-100` 을 돌려줍니다. 어떤 접근법이 더 적절한지는 응용 프로그램에 달려있습니다.

[2] x가 y의 정확한 정수배와 아주 가까우면, 라운딩(rounding) 때문에 `x//y` 는 `(x-x%y)//y` 보다 1 클 수 있습니다. 그런 경우, `divmod(x,y)[0] * y + x % y` 가 x와 아주 가깝도록 유지하기 위해, 파이썬은 뒤의 결과를 돌려줍니다.

버전 3.14에서 변경: If only one operand is a complex number, the other operand is converted to a floating-point number.

The - (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common real type.

This operation can be customized using the special *__sub__()* and *__rsub__()* methods.

버전 3.14에서 변경: If only one operand is a complex number, the other operand is converted to a floating-point number.

6.8 시프트 연산

시프트 연산은 산술 연산보다 낮은 우선순위를 갖습니다.

```
shift_expr: a_expr | shift_expr ("<<" | ">>") a_expr
```

이 연산들은 정수들을 인자로 받아들입니다. 첫 번째 인자를 두 번째 인자로 주어진 비트 수만큼 왼쪽이나 오른쪽으로 밉니다(shift).

The left shift operation can be customized using the special *__lshift__()* and *__rlshift__()* methods. The right shift operation can be customized using the special *__rshift__()* and *__rrshift__()* methods.

오른쪽으로 *n* 비트 시프트 하는 것은 `pow(2,n)` 로 정수 나눗셈하는 것으로 정의됩니다. 왼쪽으로 *n* 비트 시프트 하는 것은 `pow(2,n)` 를 곱하는 것으로 정의됩니다.

6.9 이항 비트 연산

세 개의 비트 연산은 각기 다른 우선순위를 갖습니다:

```
and_expr: shift_expr | and_expr "&" shift_expr
xor_expr: and_expr | xor_expr "^" and_expr
or_expr:  xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its arguments, which must be integers or one of them must be a custom object overriding *__and__()* or *__rand__()* special methods.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers or one of them must be a custom object overriding *__xor__()* or *__rxor__()* special methods.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers or one of them must be a custom object overriding *__or__()* or *__ror__()* special methods.

6.10 비교

C와는 달리, 파이썬에서 모든 비교 연산은 같은 우선순위를 갖는데, 산술, 시프팅, 비트 연산들보다 낮습니다. 또한, C와는 달리, `a < b < c` 와 같은 표현식이 수학에서와 같은 방식으로 해석됩니다.

```
comparison:    or_expr (comp_operator or_expr)*
comp_operator: "<" | ">" | "==" | ">=" | "<=" | "!="
               | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: `True` or `False`. Custom *rich comparison methods* may return non-boolean values. In this case Python will call `bool()` on such value in boolean contexts.

비교는 자유롭게 연결될 수 있습니다, 예를 들어, `x < y <= z` 는 `x < y and y <= z` 와 동등한데, 차이점은 `y` 의 값을 오직 한 번만 구한다는 것입니다 (하지만 두 경우 모두 `x < y` 가 거짓이면 `z` 의 값은 구하지 않습니다).

형식적으로, *a*, *b*, *c*, ⋯, *y*, *z* 가 표현식이고, *op1*, *op2*, ⋯, *opN* 가 비교 연산자면, `a op1 b op2 c ... y opN z` 는 각 표현식의 값을 최대 한 번만 구한다는 점을 제외하고는 `a op1 b and b op2 c and ... y opN z` 와 동등합니다.

`a op1 b op2 c` 가 *a* 와 *c* 간의 어떤 종류의 비교도 암시하지 않기 때문에, 예를 들어, `x < y > z` 이 완벽하게 (아마 이쁘지는 않더라도) 올바르다는 것에 주의해야 합니다.

6.10.1 값 비교

연산자 <, >, ==, >=, <=, != 는 두 객체의 값을 비교합니다. 객체들이 같은 형일 필요는 없습니다.

객체, 값, 형 장은 객체들이 (형과 아이덴티티에 더해) 값을 갖는다고 말하고 있습니다. 파이썬에서 객체의 값은 좀 추상적인 개념입니다: 예를 들어, 객체의 값에 대한 규범적인(canonical) 액세스 방법은 없습니다. 또한, 객체의 값이 특별한 방식(예를 들어, 모든 데이터 어트리뷰트로 구성되는 것)으로 구성되어야 한다는 요구 사항도 없습니다. 비교 연산자는 객체의 값이 무엇인지에 대한 특정한 종류의 개념을 구현합니다. 객체의 값을 비교를 통해 간접적으로 정의한다고 생각해도 좋습니다.

Because all types are (direct or indirect) subtypes of `object`, they inherit the default comparison behavior from `object`. Types can customize their comparison behavior by implementing *rich comparison methods* like `__lt__()`, described in 기본적인 커스터마이제이션.

동등 비교 (== 와 !=) 의 기본 동작은 객체의 아이덴티티에 기반을 둡니다. 그래서, 같은 아이덴티티를 갖는 인스턴스 간의 동등 비교는 같음을 주고, 다른 아이덴티티를 갖는 인스턴스 간의 동등 비교는 다름을 줍니다. 이 기본 동작의 동기는 모든 객체가 반사적(reflexive) (즉, `x is y` 는 `x == y` 를 암시합니다) 이도록 만들고자 하는 욕구입니다.

기본 대소 비교(order comparison) (<, >, <=, >=) 는 제공되지 않습니다; 시도하면 `TypeError` 를 일으킵니다. 이 기본 동작의 동기는 동등함과 유사한 항등 관계가 없다는 것입니다.

다른 아이덴티티를 갖는 인스턴스들이 항상 서로 다르다는, 기본 동등 비교의 동작은, 객체의 값과 값 기반의 동등함에 대한 나름의 정의를 가진 형들이 필요로 하는 것과는 크게 다를 수 있습니다. 그런 형들은 자신의 비교 동작을 커스터마이즈 할 필요가 있고, 사실 많은 내장형이 그렇게 하고 있습니다.

다음 목록은 가장 중요한 내장형들의 비교 동작을 기술합니다.

- 내장 숫자 형 ((typesnumeric)) 과 표준 라이브러리 형 `fractions.Fraction` 과 `decimal.Decimal` 에 속하는 숫자들은, 복소수가 대소 비교를 지원하지 않는다는 제약 사항만 빼고는, 같거나 다른 형들 간의 비교가 가능합니다. 관련된 형들의 한계 안에서, 정밀도의 손실 없이 수학적으로 (알고리즘적으로) 올바르게 비교합니다.

  NaN(not-a-number) 값들 `float('NaN')` 과 `decimal.Decimal('NaN')` 은 특별합니다. 모든 숫자와 NaN 간의 비교는 거짓입니다. 반 직관적으로 내포하고 있는 것은, NaN 이 자신과 같지 않다는 것입니다. 예를 들어, `x = float('NaN')`, `3 < x`, `x < 3` 및 `x == x`는 모두 거짓이지만, `x != x`는 참입니다. 이 동작은 IEEE 754를 준수합니다.

- `None` and `NotImplemented` are singletons. **PEP 8** advises that comparisons for singletons should always be done with `is` or `is not`, never the equality operators.

- 바이너리 시퀀스들 (`bytes` 나 `bytearray` 의 인스턴스들)은 형을 건너 상호 비교될 수 있습니다. 이것들은 요소들의 숫자 값을 사용해서 사전식으로(lexicographically) 비교합니다.

- 문자열들 (`str` 의 인스턴스들) 은 문자들의 유니코드 코드 포인트(Unicode code points) (내장 함수 `ord()` 의 결과)를 사용해서 사전식으로 비교합니다.[3]

  문자열과 바이너리 시퀀스는 직접 비교할 수 없습니다.

- 시퀀스들 (`tuple`, `list`, `range` 의 인스턴스들)은 같은 형끼리 비교될 수 있는데, range는 대소 비교를 지원하지 않습니다. 서로 다른 형들 간의 동등 비교는 다름을 주고, 서로 다른 형들 간의 대소 비교는 `TypeError` 를 일으킵니다.

[3] 유니코드 표준은 코드 포인트(*code points*) (예를 들어, U+0041) 와 추상 문자(*abstract characters*) (예를 들어, "LATIN CAPITAL LETTER A") 를 구분합니다. 유니코드에 있는 대부분의 추상 문자들이 오직 하나의 코드 포인트만으로 표현되지만, 추가로 하나 이상의 코드 포인트의 시퀀스로 표현될 수 있는 추상 문자들이 많이 있습니다. 예를 들어, 추상 문자 "LATIN CAPITAL LETTER C WITH CEDILLA" 는 코드 위치 U+00C7 에 있는 한 개의 복합 문자(*precomposed character*) 나 코드 위치 U+0043 (LATIN CAPITAL LETTER C) 에 있는 기본 문자(*base character*) 와 뒤따르는 코드 위치 U+0327 (COMBINING CEDILLA) 에 있는 결합 문자(*combining character*) 의 시퀀스로 표현될 수 있습니다.

문자열의 비교 연산자는 유니코드 코드 포인트 수준에서 비교합니다. 이것은 사람에게 반 직관적일 수 있습니다. 예를 들어, `"\u00C7" == "\u0043\u0327"` 는 거짓입니다, 설사 두 문자열이 같은 추상 문자 "LATIN CAPITAL LETTER C WITH CEDILLA"를 표현할지라도 그렇습니다.

문자열을 추상 문자 수준에서 비교하려면 (즉, 사람에게 직관적인 방법으로), `unicodedata.normalize()` 를 사용하십시오.

시퀀스는 대응하는 요소 간의 비교를 사용해서 사전적으로 비교합니다. 내장 컨테이너는 일반적으로 동일한(identical) 객체가 자신과 같다고(equal) 가정합니다. 이를 통해 동일한 객체에 대한 동등성(equality) 검사를 우회하여 성능을 개선하고 내부 불변성을 유지합니다.

내장 컬렉션들의 사전적인 비교는 다음과 같이 이루어집니다:

- 두 컬렉션이 같다고 비교되기 위해서는, 같은 형이고, 길이가 같고, 대응하는 요소들의 각 쌍이 같다고 비교되어야 합니다 (예를 들어, `[1,2] == (1,2)` 는 거짓인데, 형이 다르기 때문입니다).
- 대소 비교를 지원하는 컬렉션들은 첫 번째로 다른 요소들과 같은 순서를 줍니다 (예를 들어, `[1, 2,x] <= [1,2,y]` 는 `x <= y` 와 같은 값입니다). 대응하는 요소가 없는 경우 더 짧은 컬렉션이 작다고 비교됩니다 (예를 들어, `[1,2] < [1,2,3]` 은 참입니다).

- Mappings (instances of `dict`) compare equal if and only if they have equal `(key, value)` pairs. Equality comparison of the keys and values enforces reflexivity.

  대소 비교 (<, >, <=, >=) 는 `TypeError` 를 일으킵니다.

- 집합들 (`set` 이나 `frozenset` 의 인스턴스들)은 같은 형들과 서로 다른 형들 간에 비교될 수 있습니다.

  이것들은 부분집합(subset)과 상위집합(superset)을 뜻하는 대소비교 연산자들을 정의합니다. 이 관계는 전 순서(total ordering)를 정의하지 않습니다 (예를 들어, 두 집합 `{1,2}` 와 `{2,3}` 는 다르면서도, 하나가 다른 하나의 부분집합이지도, 하나가 다른 하나의 상위집합이지도 않습니다). 따라서, 전 순서에 의존하는 함수의 인자로는 적합하지 않습니다 (예를 들어, `min()`, `max()`, `sorted()` 에 입력으로 집합의 리스트를 제공하면 정의되지 않은 결과를 줍니다).

  집합의 비교는 그 요소들의 반사성을 강제합니다.

- 대부분의 다른 내장형들은 비교 메서드들을 구현하지 않기 때문에, 기본 비교 동작을 계승합니다.

비교 동작을 커스터마이즈하는 사용자 정의 클래스들은 가능하다면 몇 가지 일관성 규칙을 준수해야 합니다:

- 동등 비교는 반사적(reflexive)이어야 합니다. 다른 말로 표현하면, 아이덴티티가 같은 객체는 같다고 비교되어야 합니다:

  `x is y` 면 `x == y` 다.

- 비교는 대칭적(symmetric)이어야 합니다. 다른 말로 표현하면, 다음과 같은 표현식은 같은 결과를 주어야 합니다:

  `x == y` 와 `y == x`

  `x != y` 와 `y != x`

  `x < y` 와 `y > x`

  `x <= y` 와 `y >= x`

- 비교는 추이적(transitive)이어야 합니다. 다음 (철저하지 않은) 예들이 이것을 예증합니다:

  `x > y and y > z` 면 `x > z` 다

  `x < y and y <= z` 면 `x < z` 다

- 역 비교는 논리적 부정이 되어야 합니다. 다른 말로 표현하면, 다음 표현식들이 같은 값을 주어야 합니다:

  `x == y` 와 `not x != y`

  `x < y` 와 `not x >= y` (전 순서의 경우)

  `x > y` 와 `not x <= y` (전 순서의 경우)

  마지막 두 표현식은 전 순서 컬렉션에 적용됩니다 (예를 들어, 시퀀스에는 적용되지만, 집합과 매핑은 그렇지 않습니다). `total_ordering()` 데코레이터도 보십시오.

- `hash()` 결과는 동등성과 일관성을 유지해야 합니다. 같은 객체들은 같은 해시값을 같거나 해시 불가능으로 지정되어야 합니다.

파이썬은 이 일관성 규칙들을 강제하지 않습니다. 사실 NaN 값들은 이 규칙을 따르지 않는 예입니다.

6.10.2 멤버십 검사 연산

연산자 *in* 과 *not in* 은 멤버십을 검사합니다. `x in s` 는 *x* 가 *s* 의 멤버일 때 `True` 를, 그렇지 않을 때 `False` 를 줍니다. `x not in s` 은 `x in s` 의 부정을 줍니다. 딕셔너리 뿐만 아니라 모든 내장 시퀀스들과 집합 형들이 이것을 지원하는데, 딕셔너리의 경우는 `in` 이 딕셔너리에 주어진 키가 있는지 검사합니다. list, tuple, set, frozenset, dict, collections.deque 와 같은 컨테이너형들의 경우, 표현식 `x in y` 는 `any(x is e or x == e for e in y)` 와 동등합니다.

문자열과 바이트열 형의 경우, `x in y` 는 *x* 가 *y* 의 부분 문자열(substring)인 경우, 그리고 오직 그 경우만 `True` 입니다. 동등한 검사는 `y.find(x) != -1` 입니다. 빈 문자열은 항상 다른 문자열들의 부분 문자열로 취급되기 때문에, `"" in "abc"` 은 `True` 를 돌려줍니다.

For user-defined classes which define the *__contains__()* method, `x in y` returns `True` if `y.__contains__(x)` returns a true value, and `False` otherwise.

For user-defined classes which do not define *__contains__()* but do define *__iter__()*, `x in y` is `True` if some value `z`, for which the expression `x is z or x == z` is true, is produced while iterating over `y`. If an exception is raised during the iteration, it is as if *in* raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines *__getitem__()*, `x in y` is `True` if and only if there is a non-negative integer index *i* such that `x is y[i] or x == y[i]`, and no lower integer index raises the `IndexError` exception. (If any other exception is raised, it is as if *in* raised that exception).

연산자 *not in* 은 *in* 의 논리적 부정으로 정의됩니다.

6.10.3 아이덴티티 비교

연산자 *is* 와 *is not* 은 객체의 아이덴티티를 검사합니다: `x is y` 는 *x* 와 *y* 가 아이덴티티가 같은 객체일 때, 그리고 오직 그 경우만 참입니다. 객체의 아이덴티티는 `id()` 함수를 사용해서 결정됩니다. `x is not y` 은 논리적 부정 값을 줍니다.[4]

6.11 논리 연산(Boolean operations)

```
or_test:  and_test | or_test "or" and_test
and_test: not_test | and_test "and" not_test
not_test: comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a *__bool__()* method.

연산자 *not* 은 그 인자가 거짓이면 `True` 를, 그렇지 않으면 `False` 를 줍니다.

표현식 `x and y` 는 먼저 *x* 의 값을 구합니다; *x* 가 거짓이면 그 값을 돌려줍니다; 그렇지 않으면 *y* 의 값을 구한 후에 그 결과를 돌려줍니다.

표현식 `x or y` 는 먼저 *x* 의 값을 구합니다; *x* 가 참이면 그 값을 돌려줍니다. 그렇지 않으면 *y* 의 값을 구한 후에 그 결과를 돌려줍니다.

and 와 *or* 어느 것도 반환 값이나 그 형을 `False` 와 `True` 로 제한하지 않고, 대신 마지막에 값이 구해진 인자를 돌려줌에 주의해야 합니다. 이것은 때로 쓸모가 있습니다, 예를 들어 `s` 가 문자열이고 비어 있으면 기본값으로 대체되어야 한다면, 표현식 `s or 'foo'` 는 원하는 값을 제공합니다. *not* 은 새 값을 만들어야 하므로, 그 인자의 형과 관계없이 논리값(boolean value)을 돌려줍니다 (예를 들어, `not 'foo'` 는 `''` 가 아니라 `False` 를 만듭니다.)

[4] 자동 가비지-수거(automatic garbage-collection)와 자유 목록(free lists)과 디스크립터(descriptor)의 동적인 성격 때문에, *is* 연산자를 인스턴스 메서드들이나 상수들을 비교하는 것과 같은 특정한 방식으로 사용할 때, 겉으로 보기에 이상한 동작을 감지할 수 있습니다. 더 자세한 정보는 그들의 문서를 확인하십시오.

6.12 대입 표현식

```
assignment_expression: [identifier ":="] expression
```

An assignment expression (sometimes also called a "named expression" or "walrus") assigns an *expression* to an *identifier*, while also returning the value of the *expression*.

일반적인 사용 사례 중 하나는 일치하는 정규식을 처리할 때입니다:

```
if matching := pattern.search(data):
    do_something(matching)
```

또는, 청크로 파일 스트림을 처리할 때:

```
while chunk := file.read(9000):
    process(chunk)
```

Assignment expressions must be surrounded by parentheses when used as expression statements and when used as sub-expressions in slicing, conditional, lambda, keyword-argument, and comprehension-if expressions and in `assert`, `with`, and `assignment` statements. In all other places where they can be used, parentheses are not required, including in `if` and `while` statements.

Added in version 3.8: 대입 표현식에 대한 더 자세한 내용은 **PEP 572**를 참조하세요.

6.13 조건 표현식(Conditional expressions)

```
conditional_expression: or_test ["if" or_test "else" expression]
expression:             conditional_expression | lambda_expr
```

조건 표현식은 (때로 "삼 항 연산자(ternary operator)"라고 불립니다) 모든 파이썬 연산에서 가장 낮은 우선순위를 갖습니다.

표현식 `x if C else y`은 먼저 x 대신에 조건 C의 값을 구합니다. C가 참이면, x의 값이 구해지고 그 값을 돌려줍니다; 그렇지 않으면, y의 값을 구한 후에 그 결과를 돌려줍니다.

조건 표현식에 대한 더 자세한 내용은 **PEP 308** 를 참조하세요.

6.14 람다(Lambdas)

```
lambda_expr: "lambda" [parameter_list] ":" expression
```

람다 표현식은 (때로 람다 형식(lambda forms)이라고 불립니다) 이름 없는 함수를 만드는 데 사용됩니다. 표현식 `lambda parameters: expression`는 함수 객체를 줍니다. 이 이름 없는 객체는 이렇게 정의된 함수 객체처럼 동작합니다:

```
def <lambda>(parameters):
    return expression
```

매개변수 목록의 문법은 함수 정의 섹션을 보세요. 람다 표현식으로 만들어진 함수는 문장(statements)이나 어노테이션(annotations)을 포함할 수 없음에 주의해야 합니다.

6.15 표현식 목록(Expression lists)

```
starred_expression:       ["*"] or_expr
flexible_expression:      assignment_expression | starred_expression
flexible_expression_list: flexible_expression ("," flexible_expression)* [","]
starred_expression_list:  starred_expression ("," starred_expression)* [","]
expression_list:          expression ("," expression)* [","]
```

```
yield_list:              expression_list | starred_expression "," [starred_expression_list]
```

리스트나 집합 디스플레이의 일부일 때를 제외하고, 최소한 하나의 쉼표를 포함하는 표현식 목록은 튜플을 줍니다. 튜플의 길이는 목록에 있는 표현식의 개수입니다. 표현식들은 왼쪽에서 오른쪽으로 값이 구해집니다.

애스터리스크(asterisk) `*` 는 이터러블 언 패킹(*iterable unpacking*)을 나타냅니다. 피연산자는 반드시 이터러블 이어야 합니다. 그 이터러블이 항목들의 시퀀스로 확장되어서, 언 패킹 지점에서 새 튜플, 리스트, 집합에 포함됩니다.

Added in version 3.5: 표현식 목록에서의 이터러블 언 패킹, **PEP 448** 에서 최초로 제안되었습니다.

Added in version 3.11: Any item in an expression list may be starred. See **PEP 646**.

A trailing comma is required only to create a one-item tuple, such as `1,`; it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

6.16 값을 구하는 순서

파이썬은 왼쪽에서 오른쪽으로 표현식의 값을 구합니다. 대입의 값을 구하는 동안, 우변의 값이 좌변보다 먼저 구해짐에 주목하십시오.

다음 줄들에서, 표현식은 그들의 끝에 붙은 숫자들의 순서대로 값이 구해집니다:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

6.17 연산자 우선순위

The following table summarizes the operator precedence in Python, from highest precedence (most binding) to lowest precedence (least binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation and conditional expressions, which group from right to left).

비교, 멤버십 검사, 아이덴티티 검사들은 모두 같은 우선순위를 갖고 비교 섹션에서 설명한 것처럼 왼쪽에서 오른쪽으로 이어붙이기(chaining) 하는 기능을 갖습니다.

연산자	설명
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	결합(binding) 또는 괄호 친 표현식, 리스트 디스플레이, 딕셔너리 디스플레이, 집합 디스플레이
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	서브스크립션, 슬라이싱, 호출, 어트리뷰트 참조
`await x`	어웨이트 표현식
`**`	거듭제곱[5]
`+x`, `-x`, `~x`	양, 음, 비트 NOT
`*`, `@`, `/`, `//`, `%`	곱셈, 행렬 곱셈, 나눗셈, 정수 나눗셈, 나머지[6]
`+`, `-`	덧셈과 뺄셈
`<<`, `>>`	시프트
`&`	비트 AND
`^`	비트 XOR
`|`	비트 OR
`in`, *`not in`*, *`is`*, *`is not`*, `<`, `<=`, `>`, `>=`, `!=`, `==`	비교, 멤버십 검사와 아이덴티티 검사를 포함합니다
`not x`	논리 NOT
`and`	논리 AND
`or`	논리 OR
`if` – `else`	조건 표현식
`lambda`	람다 표현식
`:=`	대입 표현식

[5] 거듭제곱 연산자 `**` 는 오른쪽에 오는 산술이나 비트 일 항 연산자보다 약하게 결합합니다, 즉, `2**-1` 는 `0.5` 입니다.
[6] `%` 연산자는 문자열 포매팅에도 사용됩니다; 같은 우선순위가 적용됩니다.

CHAPTER 7

단순문(Simple statements)

단순문은 하나의 논리적인 줄 안에 구성됩니다. 여러 개의 단순문이 세미콜론으로 분리되어 하나의 줄에 나올 수 있습니다. 단순문의 문법은 이렇습니다:

```
simple_stmt: expression_stmt
             | assert_stmt
             | assignment_stmt
             | augmented_assignment_stmt
             | annotated_assignment_stmt
             | pass_stmt
             | del_stmt
             | return_stmt
             | yield_stmt
             | raise_stmt
             | break_stmt
             | continue_stmt
             | import_stmt
             | future_stmt
             | global_stmt
             | nonlocal_stmt
             | type_stmt
```

7.1 표현식 문

표현식 문은 값을 계산하고 출력하거나, (보통) 프로시저(procedure) (의미 없는 결과를 돌려주는 함수; 파이썬에서 프로시저는 `None` 값을 돌려줍니다)를 호출하기 위해 (대부분 대화형으로) 사용됩니다. 표현식 문의 다른 사용도 허락되고 때때로 쓸모가 있습니다.

```
expression_stmt: starred_expression
```

표현식 문은 (하나의 표현식일 수 있는) 표현식 목록의 값을 구합니다.

대화형 모드에서, 값이 `None` 이 아니면, 내장 `repr()` 함수를 사용해 문자열로 변환되고, 그렇게 나온 문자열을 별도의 줄에 표준 출력으로 보냅니다 (결과가 `None` 일 때는 그렇지 않아서, 프로시저 호출은 어떤 출력도 만들지 않습니다.),

7.2 대입문

대입문은 이름을 값에 (재)연결하고 가변 객체의 어트리뷰트나 항목들을 수정합니다.

```
assignment_stmt: (target_list "=")+ (starred_expression | yield_expression)
target_list:     target ("," target)* [","]
target:          identifier
                 | "(" [target_list] ")"
                 | "[" [target_list] "]"
                 | attributeref
                 | subscription
                 | slicing
                 | "*" target
```

(*attributeref*, *subscription*, *slicing* 의 문법 정의는 프라이머리 섹션을 보십시오.)

대입문은 표현식 목록 (이것이 하나의 표현식일 수도, 쉼표로 분리된 목록일 수도 있는데, 후자의 경우는 튜플이 만들어진다는 것을 기억하십시오) 의 값을 구하고, 왼쪽에서 오른쪽으로, 하나의 결과 객체를 타깃 목록의 각각에 대입합니다.

대입은 타깃 (목록)의 형태에 따라 재귀적으로 정의됩니다. 타깃이 가변 객체의 일부 (어트리뷰트 참조나 서브스크립션이나 슬라이싱) 면, 가변 객체가 최종적으로 대입을 수행해야만 하고, 그것이 올바른지 아닌지를 결정하고, 대입이 받아들여 질 수 없으면 예외를 일으킬 수 있습니다. 다양한 형들이 주시하는 규칙들과 발생하는 예외들은 그 객체 형의 정의에서 주어진다 (표준형 계층 섹션을 보십시오).

객체를 타깃 목록, 괄호나 대괄호로 둘러싸일 수 있는데 생략할 수 있습니다, 에 대입하는 것은 다음과 같이 재귀적으로 정의됩니다.

- 타깃 목록이 (선택적으로 괄호에 들어있는) 뒤따르는 쉼표가 없는 하나의 타깃이면 객체는 타깃에 대입됩니다.
- Else:
  - 타깃 목록이 애스터리스크(asterisk)를 앞에 붙인 타깃, "스타드(starred)" 타깃이라고 불립니다, 하나를 포함하면: 객체는 적어도 타깃 목록에 나오는 타깃의 수보다 하나 작은 개수의 항목을 제공하는 이터러블이어야 합니다. 이터러블의 처음 항목들은, 왼쪽에서 오른쪽으로, 스타드 타깃 앞에 나오는 타깃들에 대입됩니다. 이터러블의 마지막 항목들은 스타드 타깃 뒤에 나오는 타깃들에 대입됩니다. 이터러블의 나머지 항목들로 구성된 리스트가 스타드 타깃에 대입됩니다 (이 리스트는 비어있을 수 있습니다).
  - 그렇지 않으면: 객체는 타깃 목록에 나오는 타깃의 수와 같은 수의 항목들을 제공하는 이터러블이어야 하고, 항목들은, 왼쪽에서 오른쪽으로, 대응하는 타깃들에 대입됩니다.

하나의 타깃에 대한 객체의 대입은 다음과 같이 재귀적으로 정의됩니다.

- 타깃이 식별자 (이름) 면:
  - 그 이름이 현재 코드 블록에 있는 *global* 나 *nonlocal* 문에 등장하지 않으면: 그 이름은 현재 지역 이름 공간에서 객체에 연결됩니다.
  - 그렇지 않으면: 그 이름은 각각 전역 이름 공간이나 *nonlocal* 에 의해 결정되는 외부 이름 공간에서 객체에 연결됩니다.

  그 이름이 이미 연결되어 있으면 재연결됩니다. 이것은 기존에 연결되어 있던 객체의 참조 횟수가 0 이 되도록 만들어서, 객체가 점유하던 메모리가 반납되고 파괴자(destructor) (갖고 있다면) 가 호출되도록 만들 수 있습니다.
- 타깃이 어트리뷰트 참조면: 참조의 프라이머리 표현식의 값을 구합니다. 이것은 대입 가능한 어트리뷰트를 가진 객체를 주어야 하는데, 그렇지 않으면 `TypeError` 가 일어납니다. 그에 그 객체에 주어진 어트리뷰트로 객체를 대입하도록 요청합니다; 대입을 수행할 수 없다면 예외 (보통 `AttributeError` 이지만, 꼭 그럴 필요는 없다) 를 일으킵니다.

  주의 사항: 객체가 클래스 인스턴스이고 어트리뷰트 참조가 대입 연산자의 양쪽에서 모두 등장하면, 우변(right-hand side) 표현식, `a.x` 는 인스턴스 어트리뷰트나 (인스턴스 어트리뷰트가 없다면) 클래스 어트리뷰트를 액세스할 수 있습니다. 좌변(left-hand side) 타깃 `a.x` 는 항상 필요하면 만들어서라도

항상 인스턴스 어트리뷰트를 설정합니다. 그래서, 두 `a.x` 가 같은 어트리뷰트를 가리키는 것은 필요조건이 아닙니다: 우변 표현식이 클래스 어트리뷰트를 가리킨다면, 좌변은 대입의 타깃으로 새 인스턴스 어트리뷰트를 만듭니다:

```
class Cls:
    x = 3             # class variable
inst = Cls()
inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as 3
```

이 설명이 `property()` 로 만들어진 프로퍼티(property)와 같은 디스크립터 어트리뷰트에 적용될 필요는 없습니다.

- 타깃이 서브스크립션이면: 참조에 있는 프라이머리 표현식의 값을 구합니다. (리스트 같은) 가변 시퀀스 객체나 (딕셔너리 같은) 매핑 객체가 나와야 합니다. 그런 다음, 서브 스크립트 표현식의 값을 구합니다.

  프라이머리가 (리스트 같은) 가변 시퀀스 객체면, 서브 스크립트는 정수가 나와야 합니다. 음수면, 시퀀스의 길이가 더해집니다. 결괏값은 시퀀스의 길이보다 작은 음이 아닌 정수여야 하고, 시퀀스에 그 인덱스를 가진 항목에 객체를 대입하라고 요청합니다. 인덱스가 범위를 벗어나면, `IndexError` 를 일으킵니다 (서브 스크립트 된 시퀀스에 대한 대입은 리스트에 새 항목을 추가할 수 없습니다).

  If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping' s key type, and the mapping is then asked to create a key/value pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

  For user-defined objects, the *`__setitem__()`* method is called with appropriate arguments.

- 타깃이 슬라이싱이면: 참조의 프라이머리 표현식의 값을 구합니다. (리스트 같은) 가변 시퀀스 객체가 나와야 합니다. 대입되는 객체는 같은 형의 시퀀스 객체야 합니다. 그런 다음, 존재한다면 하한과 상한 표현식의 값을 구합니다; 기본값은 0과 시퀀스의 길이다. 경곗값은 정수가 되어야 합니다. 둘 중 어느 것이건 음수가 나오면, 시퀀스의 길이를 더합니다. 그렇게 얻어진 경곗값들을 0과 시퀀스의 길이나 그 사이에 들어가는 값이 되도록 자릅니다. 마지막으로 시퀀스 객체에 슬라이스를 대입되는 시퀀스로 변경하도록 요청합니다. 타깃 시퀀스가 허락한다면, 슬라이스의 길이는 대입되는 시퀀스의 길이와 다를 수 있습니다.

현재 구현에서, 타깃의 문법은 표현식과 같게 유지되고, 잘못된 문법은 코드 생성 단계에서 거부되기 때문에 에러 메시지가 덜 상세해지는 결과를 낳고 있습니다.

설사 대입의 정의가 좌변과 우변 간의 중첩이 '동시적(simultaneous)' 임을 (예를 들어, `a, b = b, a` 는 두 변수를 교환합니다) 암시해도, 대입되는 변수들의 컬렉션 안에서의 중첩은 왼쪽에서 오른쪽으로 일어나서, 때로 혼동할 수 있는 결과를 낳습니다. 예를 들어, 다음과 같은 프로그램은 `[0, 2]` 를 인쇄합니다:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # i is updated, then x[i] is updated
print(x)
```

더 보기

PEP 3132 - 확장 이터러블 언 패킹
`*target` 기능에 대한 규격

7.2.1 증분 대입문(Augmented assignment statements)

증분 대입문은 한 문장에서 이항 연산과 대입문을 합치는 것입니다:

```
augmented_assignment_stmt: augtarget augop (expression_list | yield_expression)
augtarget:                 identifier | attributeref | subscription | slicing
augop:                     "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
```

```
                    | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(마지막 세 기호의 문법 정의는 프라이머리 섹션을 보십시오.)

증분 대입은 타깃 (일반 대입문과는 달리 언 패킹이 될 수 없습니다) 과 표현식 목록의 값을 구하고, 둘을 피연산자로 삼아 대입의 형에 맞는 이항 연산을 수행한 후, 원래의 타깃에 그 결과를 대입합니다. 타깃은 오직 한 번만 값이 구해집니다.

An augmented assignment statement like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

일반 대입과는 달리, 증분 대입은 우변의 값을 구하기 이전에 좌변의 값을 구합니다. 예를 들어, `a[i] += f(x)` 는 처음에 `a[i]` 를 조회한 다음, `f(x)` 의 값을 구하고, 덧셈을 수행하고, 마지막으로 그 결과를 `a[i]` 에 다시 씁니다.

하나의 문장에서 튜플과 다중 타깃으로 대입하는 것을 예외로 하면, 증분 대입문에 의한 대입은 일반 대입과 같은 방법으로 처리됩니다. 마찬가지로, 제자리 동작의 가능성을 예외로 하면, 증분 대입 때문에 수행되는 이진 연산은 일반 이진 연산과 같습니다.

어트리뷰트 참조인 타깃의 경우, 일반 대입처럼 클래스와 인스턴스 어트리뷰트에 관한 경고 가 적용됩니다.

7.2.2 어노테이트된 대입문(Annotated assignment statements)

어노테이션 대입은, 한 문장에서, 변수나 어트리뷰트 어노테이션과 생략할 수 있는 대입문을 합치는 것입니다.

```
annotated_assignment_stmt: augtarget ":" expression
                           ["=" (starred_expression | yield_expression)]
```

The difference from normal 대입문 is that only a single target is allowed.

The assignment target is considered "simple" if it consists of a single name that is not enclosed in parentheses. For simple assignment targets, if in class or module scope, the annotations are gathered in a lazily evaluated *annotation scope*. The annotations can be evaluated using the `__annotations__` attribute of a class or module, or using the facilities in the `annotationlib` module.

If the assignment target is not simple (an attribute, subscript node, or parenthesized name), the annotation is never evaluated.

이름이 함수 스코프에서 어노테이트되면, 이 이름은 그 스코프에 지역적(local)입니다. 함수 스코프에서 어노테이션은 값이 구해지거나 저장되지 않습니다.

If the right hand side is present, an annotated assignment performs the actual assignment as if there was no annotation present. If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last `__setitem__()` or `__setattr__()` call.

더 보기

PEP 526 - 변수 어노테이션 문법
주석을 통해 표현하는 대신, 변수(클래스 변수와 인스턴스 변수 포함)의 형을 어노테이트 하는 문법을 추가하는 제안.

PEP 484 - 형 힌트
정적 분석 도구와 IDE에서 사용할 수 있는 형 어노테이션에 대한 표준 문법을 제공하기 위해 `typing` 모듈을 추가하는 제안.

버전 3.8에서 변경: Now annotated assignments allow the same expressions in the right hand side as regular assignments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

버전 3.14에서 변경: Annotations are now lazily evaluated in a separate *annotation scope*. If the assignment target is not simple, annotations are never evaluated.

7.3 assert 문

assert 문은 프로그램에 디버깅 어서션(debugging assertion)을 삽입하는 편리한 방법입니다:

```
assert_stmt: "assert" expression ["," expression]
```

간단한 형태, `assert expression` 은 다음과 동등합니다

```
if __debug__:
    if not expression: raise AssertionError
```

확장된 형태, `assert expression1, expression2` 는 다음과 동등합니다

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an *assert* statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

`__debug__` 에 대한 대입은 허락되지 않습니다. 이 내장 변수의 값은 인터프리터가 시작할 때 결정됩니다.

7.4 pass 문

```
pass_stmt: "pass"
```

pass 는 널(null) 연산입니다 — 실행될 때, 아무런 일도 일어나지 않습니다. 문법적으로 문장이 필요하기는 하지만 할 일은 없을 때, 자리를 채우는 용도로 쓸모가 있습니다, 예를 들어:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

7.5 del 문

```
del_stmt: "del" target_list
```

삭제는 대입이 정의된 방식과 아주 비슷하게 재귀적으로 정의됩니다. 전체 세부 사항들을 나열하는 대신, 여기 몇 가지 힌트가 있습니다.

타깃 목록의 삭제는 각 타깃을 왼쪽에서 오른쪽으로 재귀적으로 삭제합니다.

이름의 삭제는 같은 코드 블록에 있는 *global* 문에 그 이름이 등장하는지에 따라 지역이나 전역 이름 공간에서 이름의 연결을 제거합니다. 이름이 연결되어 있지 않으면, `NameError` 예외가 일어납니다.

어트리뷰트 참조, 서브스크립션, 슬라이싱의 삭제는 관련된 프라이머리 객체로 전달됩니다; 슬라이싱의 삭제는 일반적으로 우변 형의 빈 슬라이스를 대입하는 것과 동등합니다 (하지만 이것조차 슬라이싱 되는 객체가 판단합니다).

버전 3.2에서 변경: 예전에는 이름이 중첩된 블록에서 자유 변수로 등장하는 경우 지역 이름 공간에서 삭제하는 것이 허락되지 않았습니다.

7.6 return 문

```
return_stmt: "return" [expression_list]
```

return 은 문법적으로 클래스 정의에 중첩된 경우가 아니라, 함수 정의에만 중첩되어 나타날 수 있습니다.

표현식 목록이 있으면 값을 구하고, 그렇지 않으면 None 으로 치환됩니다.

return 은 표현식 목록 (또는 None)을 반환 값으로 해서, 현재의 함수 호출을 떠납니다.

return 이 finally 절을 가진 try 문에서 제어가 벗어나도록 만드는 경우, 함수로부터 진짜로 벗어나기 전에 그 finally 절이 실행됩니다.

제너레이터 함수에서, return 문은 제너레이터가 끝났음을 가리키고, StopIteration 예외를 일으킵니다. return 문에 제공되는 값은 (있다면) StopIteration 의 생성자에 인자로 전달되어 StopIteration.value 어트리뷰트가 됩니다.

비동기 제너레이터 함수에서, 빈 return 문은 비동기 제너레이터가 끝났음을 알리고, StopAsyncIteration 예외를 일으킵니다. 비동기 제너레이터 함수에서, 비어있지 않은 return 은 문법 에러입니다.

7.7 yield 문

```
yield_stmt: yield_expression
```

A yield statement is semantically equivalent to a *yield expression*. The yield statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

```
yield <expr>
yield from <expr>
```

은 다음과 같은 yield 표현식 문장들과 동등합니다

```
(yield <expr>)
(yield from <expr>)
```

Yield expressions and statements are only used when defining a *generator* function, and are only used in the body of the generator function. Using yield in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

yield 의 뜻에 대한 전체 세부 사항들은 일드 표현식 (*Yield expressions*) 섹션을 참고하면 됩니다.

7.8 raise 문

```
raise_stmt: "raise" [expression ["from" expression]]
```

If no expressions are present, raise re-raises the exception that is currently being handled, which is also known as the *active exception*. If there isn' t currently an active exception, a RuntimeError exception is raised indicating that this is an error.

그렇지 않으면, raise 는 예외 객체로, 첫 번째 표현식의 값을 구합니다. BaseException 의 서브 클래스나 인스턴스여야 합니다. 클래스면, 예외 인스턴스는 필요할 때 인자 없이 클래스의 인스턴스를 만들어서 사용됩니다.

예외의 형 (*type*)은 예외 인스턴스의 클래스고, 값 (*value*)은 인스턴스 자신입니다.

A traceback object is normally created automatically when an exception is raised and attached to it as the __traceback__ attribute. You can create an exception and set your own traceback in one step using the with_traceback() exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The from clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the __cause__ attribute (which is writable). If the expression is an exception class, the class will be instantiated and

the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened") from exc
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an *except* or *finally* clause, or a *with* statement, is used. The previous exception is then attached as the new exception' s `__context__` attribute:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened")
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

예외에 대한 더 많은 정보를 예외 섹션에서 발견할 수 있고, 예외를 처리하는 것에 대한 정보는 *try* 문 섹션에 있습니다.

버전 3.3에서 변경: 이제 `raise X from Y` 에서 `Y` 로 `None` 이 허락됩니다.

Added the `__suppress_context__` attribute to suppress automatic display of the exception context.

버전 3.11에서 변경: If the traceback of the active exception is modified in an *except* clause, a subsequent `raise` statement re-raises the exception with the modified traceback. Previously, the exception was re-raised with the traceback it had when it was caught.

7.9 break 문

```
break_stmt: "break"
```

break 는 문법적으로 *for* 나 *while* 루프에 중첩되어서만 나타날 수 있습니다. 하지만 그 루프 안의 함수나 클래스 정의에 중첩되지는 않습니다.

가장 가까이서 둘러싸고 있는 루프를 종료하고, 그 루프가 `else` 절을 갖고 있다면 건너뜁니다(skip).

for 루프가 *break* 로 종료되면, 루프 제어 타깃은 현재값을 유지합니다.

break 가 *finally* 절을 가 *try* 문에서 제어가 벗어나도록 만드는 경우, 루프로부터 진짜로 벗어나기 전에 그 `finally` 절이 실행됩니다.

7.10 continue 문

```
continue_stmt: "continue"
```

continue 는 문법적으로 *for* 나 *while* 루프에 중첩되어서만 나타날 수 있습니다. 하지만 그 루프 안의 함수나 클래스 정의에 중첩되지는 않습니다. 가장 가까이서 둘러싸고 있는 루프가 다음 사이클로 넘어가도록 만듭니다.

continue 가 *finally* 절을 가진 *try* 문에서 제어가 벗어나도록 만드는 경우, 다음 루트 사이클을 시작하기 전에 그 `finally` 절이 실행됩니다.

7.11 임포트(import) 문

```
import_stmt:     "import" module ["as" identifier] ("," module ["as" identifier])*
                 | "from" relative_module "import" identifier ["as" identifier]
                 ("," identifier ["as" identifier])*
                 | "from" relative_module "import" "(" identifier ["as" identifier]
                 ("," identifier ["as" identifier])* [","] ")"
                 | "from" relative_module "import" "*"
module:          (identifier ".")* identifier
relative_module: "."* module | "."+
```

(*from* 절이 없는) 기본 임포트 문은 두 단계로 실행됩니다:

1. 모듈을 찾고, 로드하고, 필요하면 초기화합니다
2. 임포트(*import*) 문이 등장한 스코프의 지역 이름 공간에 이름이나 이름들을 정의합니다.

문장이 (쉼표로 분리된) 여러 개의 절을 포함하면, 마치 각 절이 별도의 임포트 문에 의해 분리된 것처럼, 두 단계는 절마다 별도로 수행됩니다.

The details of the first step, finding and loading modules, are described in greater detail in the section on the *import system*, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, *or* that an error occurred while initializing the module, which includes execution of the module's code.

요청된 모듈이 성공적으로 읽어 들여지면, 세 가지 중 한 방법으로 지역 이름 공간에 소개됩니다:

- 모듈 이름 뒤에 `as` 가 오면, `as` 뒤에 오는 이름이 임포트된 모듈에 직접 연결됩니다.
- 다른 이름이 지정되지 않고, 임포트되는 모듈이 최상위 모듈이면, 모듈의 이름이 임포트되는 모듈에 대한 참조로 지역 이름 공간에 연결됩니다.

- 임포트되는 모듈이 최상이 모듈이 아니 라면, 그 모듈을 포함하는 최상위 패키지의 이름이 최상위 패키지에 대한 참조로 지역 이름 공간에 연결됩니다. 임포트된 모듈은 직접적이기보다는 완전히 정규화된 이름(full qualified name)을 통해 액세스 되어야 합니다.

from 형은 약간 더 복잡한 절차를 사용합니다:

1. *from* 절에 지정된 모듈을 찾고, 로드하고, 필요하면 초기화합니다
2. *import* 절에 지정된 식별자들 각각에 대해:
    1. 임포트된 모듈이 그 이름의 어트리뷰트를 가졌는지 검사합니다
    2. 없으면, 그 이름의 서브 모듈을 임포트하는 것을 시도한 다음 임포트된 모듈에서 그 어트리뷰트를 다시 검사합니다
    3. 어트리뷰트가 발견되지 않으면 `ImportError` 를 일으킵니다.
    4. 그렇지 않으면, 그 값에 대한 참조가 지역 이름 공간에 저장되는데, `as` 절이 존재하면 거기에서 지정된 이름을 사용하고, 그렇지 않으면 어트리뷰트 이름을 사용합니다

사용 예:

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo, foo.bar, and foo.bar.baz imported, foo bound␣
↪locally
import foo.bar.baz as fbb  # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz␣
↪bound as fbb
from foo.bar import baz    # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz␣
↪bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

식별자들의 목록을 스타(`'*'`)로 바꾸면, 모듈에 정의된 모든 공개 이름들이 *import* 문이 등장한 스코프의 지역 이름 공간에 연결됩니다.

모듈에 정의된 공개 이름(*public names*) 은 모듈의 이름 공간에서 `__all__` 이라는 이름의 변수를 검사해서 결정됩니다; 정의되어 있다면, 문자열의 시퀀스여야 하는데, 그 모듈이 정의하거나 임포트하는 이름들입니다. `__all__` 에서 지정한 이름들은 모두 공개로 취급되고 반드시 존재해야 합니다. `__all__` 이 정의되지 않으면, 모듈의 이름 공간에서 발견되는 이름 중, 밑줄 문자(`'_'`)로 시작하지 않는 모든 이름이 공개로 취급됩니다. `__all__` 는 공개 API 전체를 포함해야 합니다. 이것의 목적은 의도치 않게 API 일부가 아닌 항목들을 노출하는 것을 방지하는 것입니다 (가령 그 모듈이 임포트하고 사용하는 라이브러리 모듈).

임포트의 와일드카드 형태 — `from module import *` — 는 모듈 수준에서만 허락됩니다. 클래스나 함수 정의에서 사용하려는 시도는 `SyntaxError` 를 일으킵니다.

임포트할 모듈을 지정할 때 모듈의 절대 이름(absolute name)을 지정할 필요는 없습니다. 모듈이나 패키지가 다른 패키지 안에 포함될 때, 같은 상위 패키지 내에서는 그 패키지 이름을 언급할 필요 없이 상대 임포트(relative import)를 할 수 있습니다. *from* 뒤에 지정되는 패키지나 모듈 앞에 붙이는 점으로, 정확한 이름을 지정하지 않고도 현재 패키지 계층을 얼마나 거슬러 올라가야 하는지 지정할 수 있습니다. 하나의 점은 이 임포트를 하는 모듈이 존재하는 현재 패키지를 뜻합니다. 두 개의 점은 한 패키지 수준을 거슬러 올라가는 것을 뜻합니다. 세 개의 점은 두 개의 수준을, 등등입니다. 그래서 `pkg` 패키지에 있는 모듈에서 `from . import mod` 를 실행하면, `pkg.mod` 를 임포트하게 됩니다. `pkg.subpkg1` 안에서 `from ..subpkg2 import mod` 를 실행하면 `pkg.subpkg2.mod` 를 임포트하게 됩니다. 상대 임포트에 대한 규격은 패키지 상대 임포트 절에 들어있습니다.

로드할 모듈들을 동적으로 결정하는 응용 프로그램들을 지원하기 위해 `importlib.import_module()` 이 제공됩니다.

인자 `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`로 감사 이벤트 `import` 를 발생시킵니다.

7.11.1 퓨처 문

퓨처 문(*future statement*)은 컴파일러가 특정한 모듈을 특별한 문법이나 개념을 사용해서 컴파일하도록 만드는 지시어(directive)인데, 그 기능은 미래에 출시되는 파이썬에서 표준이 되는 것입니다.

퓨처 문의 목적은 언어에 호환되지 않는 변경이 도입된 미래 버전의 파이썬으로 옮겨가는 것을 쉽게 만드는 것입니다. 그 기능이 표준이 되는 배포 이전에 모듈 단위로 새 기능을 사용할 수 있도록 만듭니다.

```
future_stmt: "from" "__future__" "import" feature ["as" identifier]
             ("," feature ["as" identifier])*
             | "from" "__future__" "import" "(" feature ["as" identifier]
             ("," feature ["as" identifier])* [","] ")"
feature:     identifier
```

퓨처 문은 모듈의 거의 처음에 나와야 합니다. 퓨처 문 앞에 나올 수 있는 줄들은:

- 모듈 독스트링(docstring) (있다면),
- 주석
- 빈 줄, 그리고
- 다른 퓨처 문들

퓨처 문을 사용해야 하는 유일한 기능은 `annotations` 입니다 (**PEP 563**을 참조하십시오).

과거에 퓨처 문을 통해 활성화되던 기능들은 여전히 파이썬 3에 의해 인식됩니다. 이 목록에는 `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` 및 `with_statement` 가 포함됩니다. 이것들은 잉여물인데 항상 활성화되고, 오직 과거 호환성을 위해 유지되고 있기 때문입니다.

퓨처 문은 구체적으로는 컴파일 시점에 인식되고 다뤄집니다: 핵심 구성물들의 의미에 대한 변경은 종종 다른 코드 생성을 통해 구현됩니다. 새 기능이 호환되지 않는 (새로운 예약어처럼) 새로운 문법을 도입하는 경우조차 가능한데, 이 경우는 컴파일러가 모듈을 다르게 파싱할 수 있습니다. 그런 결정들은 실행 시점으로 미뤄질 수 없습니다..

배포마다, 컴파일러는 어떤 기능 이름들이 정의되어 있는지 알고, 만약 퓨처 문이 알지 못하는 기능을 포함하고 있으면 컴파일 시점 에러를 일으킵니다.

직접적인 실행 시점의 개념은 다른 임포트 문들과 같습니다: 표준 모듈 `__future__`, 후에 설명합니다, 다 있고, 퓨처 문이 실행되는 시점에 일반적인 방법으로 임포트됩니다.

흥미로운 실행 시점의 개념들은 퓨처 문에 의해 활성화되는 구체적인 기능들에 달려있습니다.

이런 문장에는 아무것도 특별한 것이 없음에 주의해야 합니다:

```
import __future__ [as name]
```

이것은 퓨처 문이 아닙니다; 아무런 특별한 개념이나 문법적인 제약이 없는 평범한 임포트 문일 뿐입니다.

Code compiled by calls to the built-in functions `exec()` and `compile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to `compile()` — see the documentation of that function for details.

대화형 인터프리터 프롬프트에서 입력된 퓨처 문은 인터프리터 세션의 남은 기간 효과를 발생시킵니다. 인터프리터가 `-i`, 실행할 스크립트 이름이 전달됩니다, 옵션으로 시작하고, 그 스크립트가 퓨처 문을 포함하면, 스크립트가 실행된 이후에 시작되는 대화형 세션에서도 효과를 유지합니다.

더 보기

PEP 236 - 백 투 더 __future__
__future__ 메커니즘에 대한 최초의 제안.

7.12 global 문

```
global_stmt: "global" identifier ("," identifier)*
```

The *global* statement causes the listed identifiers to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

The *global* statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its global declaration in the scope.

프로그래머의 주의 사항: *global* 은 파서에 주는 지시자(directive)입니다. `global` 문과 같은 시점에 파싱되는 코드에만 적용됩니다. 특히, 내장 `exec()` 함수로 공급되는 문자열이나 코드 객체에 포함된 `global` 문은 그 함수 호출을 포함하는 코드 블록에는 영향을 주지 않고, 그런 문자열에 포함된 코드 역시 함수 호출을 포함하는 코드에 있는 `global` 문에 영향을 받지 않습니다. `eval()` 과 `compile()` 함수들도 마찬가지입니다.

7.13 nonlocal 문

```
nonlocal_stmt: "nonlocal" identifier ("," identifier)*
```

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its nonlocal scopes are the local scopes of the enclosing functions. The *nonlocal* statement causes the listed identifiers to refer to names previously bound in nonlocal scopes. It allows encapsulated code to rebind such nonlocal identifiers. If a name is bound in more than one nonlocal scope, the nearest binding is used. If a name is not bound in any nonlocal scope, or if there is no nonlocal scope, a `SyntaxError` is raised.

The *nonlocal* statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its nonlocal declaration in the scope.

더 보기

PEP 3104 - 바깥 스코프에 있는 이름들에 대한 액세스
nonlocal 문의 규격.

Programmer' s note: *nonlocal* is a directive to the parser and applies only to code parsed along with it. See the note for the *global* statement.

7.14 The type statement

```
type_stmt: 'type' identifier [type_params] "=" expression
```

The `type` statement declares a type alias, which is an instance of `typing.TypeAliasType`.

For example, the following statement creates a type alias:

```
type Point = tuple[float, float]
```

This code is roughly equivalent to:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

`annotation-def` indicates an *annotation scope*, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias' s `__value__` attribute (see *Lazy evaluation*). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a *type parameter list* after the name. See *Generic type aliases* for more.

`type` is a *soft keyword*.

Added in version 3.12.

더 보기

PEP 695 - Type Parameter Syntax
Introduced the `type` statement and syntax for generic classes and functions.

CHAPTER 8

복합문(Compound statements)

복합문은 다른 문장들(의 그룹들)을 포함합니다; 어떤 방법으로 그 다른 문장들의 실행에 영향을 주거나 제어합니다. 간단하게 표현할 때, 전체 복합문을 한 줄로 쓸 수 있기는 하지만, 일반적으로 복합문은 여러 줄에 걸칩니다.

if, *while*, *for* 문장은 전통적인 제어 흐름 구조를 구현합니다. 문장들의 그룹에 대해 *try* 는 예외 처리기나 정리(cleanup) 코드 또는 그 둘 모두를 지정하는 반면, *with* 문은 코드 블록 주변으로 초기화와 파이널리제이션 코드를 실행할 수 있도록 합니다. 함수와 클래스 정의 또한 문법적으로 복합문입니다.

복합문은 하나나 그 이상의 '절' 로 구성됩니다. 절은 헤더와 '스위트(suite)' 로 구성됩니다. 특정 복합문의 절 헤더들은 모두 같은 들여쓰기 수준을 갖습니다. 각 절 헤더는 특별하게 식별되는 키워드로 시작하고 콜론으로 끝납니다. 스위트는 절에 의해 제어되는 문장들의 그룹입니다. 스위트는 헤더의 콜론 뒤에서 같은 줄에 세미콜론으로 분리된 하나나 그 이상의 단순문일 수 있습니다. 또는 그다음 줄에 들여쓰기 된 하나나 그 이상의 문장들일 수도 있습니다. 오직 후자의 형태만 중첩된 복합문을 포함할 수 있습니다; 다음과 같은 것은 올바르지 않은데, 대체로 뒤따르는 *else* 절이 있다면 어떤 *if* 절에 속하는지 명확하지 않기 때문입니다.

```
if test1: if test2: print(x)
```

또한, 이 문맥에서 세미콜론이 콜론보다 더 강하게 결합해서, 다음과 같은 예에서, `print()` 호출들은 모두 실행되거나 어느 하나도 실행되지 않습니다는 것에 주의해야 합니다:

```
if x < y < z: print(x); print(y); print(z)
```

요약하면:

```
compound_stmt: if_stmt
               | while_stmt
               | for_stmt
               | try_stmt
               | with_stmt
               | match_stmt
               | funcdef
               | classdef
               | async_with_stmt
               | async_for_stmt
               | async_funcdef
suite:         stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement:     stmt_list NEWLINE | compound_stmt
```

```
stmt_list:     simple_stmt (";" simple_stmt)* [";"]
```

문장들이 항상 NEWLINE 으로 끝나고 DEDENT 가 그 뒤를 따를 수 있음에 주목해야 합니다. 또한, 생략 가능한 연결 절들이 항상 문장을 시작시킬 수 없는 키워드로 시작하기 때문에, 모호함이 없다는 것도 주목하세요 (파이썬에서는 중첩된 *if* 문이 들여쓰기 되는 것을 요구함으로써 '매달린(dangling) *else*' 문제를 해결합니다).

명확함을 위해 다음에 오는 절들에서 나오는 문법 규칙들은 각 절을 별도의 줄에 놓도록 포매팅합니다.

8.1 if 문

if 문은 조건부 실행에 사용됩니다:

```
if_stmt: "if" assignment_expression ":" suite
         ("elif" assignment_expression ":" suite)*
         ["else" ":" suite]
```

참이 되는 것을 발견할 때까지 표현식들의 값을 하나씩 차례대로 구해서 정확히 하나의 스위트를 선택합니다 (참과 거짓의 정의는 논리 연산(*Boolean operations*) 섹션을 보세요); 그런 다음 그 스위트를 실행합니다 (그리고는 *if* 문의 다른 어떤 부분도 실행되거나 값이 구해지지 않습니다). 모든 표현식들이 거짓이면 *else* 절의 스위트가 (있다면) 실행됩니다.

8.2 while 문

while 문은 표현식이 참인 동안 실행을 반복하는 데 사용됩니다:

```
while_stmt: "while" assignment_expression ":" suite
            ["else" ":" suite]
```

이것은 표현식을 반복적으로 검사하고, 참이면, 첫 번째 스위트를 실행합니다; 표현식이 거짓이면 (처음부터 거짓일 수도 있습니다) else 절의 스위트가 (있다면) 실행되고 루프를 종료합니다.

첫 번째 스위트에서 실행되는 *break* 문은 else 절을 실행하지 않고 루프를 종료합니다. 첫 번째 스위트에서 실행되는 *continue* 문은 스위트의 나머지 부분을 건너뛰고 표현식의 검사로 돌아갑니다.

8.3 for 문

for 문은 (문자열, 튜플, 리스트 같은) 시퀀스 나 다른 이터러블 객체의 요소들을 이터레이트하는데 사용됩니다:

```
for_stmt: "for" target_list "in" starred_list ":" suite
          ["else" ":" suite]
```

The starred_list expression is evaluated once; it should yield an *iterable* object. An *iterator* is created for that iterable. The first item provided by the iterator is then assigned to the target list using the standard rules for assignments (see 대입문), and the suite is executed. This repeats for each item provided by the iterator. When the iterator is exhausted, the suite in the else clause, if present, is executed, and the loop terminates.

첫 번째 스위트에서 실행되는 *break* 문은 else 절을 실행하지 않고 루프를 종료합니다. 첫 번째 스위트에서 실행되는 *continue* 문은 스위트의 나머지 부분을 건너뛰고 다음 항목으로 넘어가거나, 다음 항목이 없으면 else 절로 갑니다.

for-루프는 타깃 목록의 변수들에 대입합니다. for-루프의 스위트에서 이루어진 것들도 포함해서, 그 변수에 앞서 대입된 값들을 모두 덮어씁니다:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in type `range()` represents immutable arithmetic sequences of integers. For instance, iterating `range(3)` successively yields 0, 1, and then 2.

버전 3.11에서 변경: Starred elements are now allowed in the expression list.

8.4 `try` 문

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt:  try1_stmt | try2_stmt | try3_stmt
try1_stmt: "try" ":" suite
           ("except" [expression ["as" identifier]] ":" suite)+
           ["else" ":" suite]
           ["finally" ":" suite]
try2_stmt: "try" ":" suite
           ("except" "*" expression ["as" identifier] ":" suite)+
           ["else" ":" suite]
           ["finally" ":" suite]
try3_stmt: "try" ":" suite
           "finally" ":" suite
```

예외에 관한 추가의 정보는 예외 섹션에서 찾을 수 있고, 예외를 일으키기 위해 *raise* 문을 사용하는 것에 관한 정보는 *raise* 문 섹션에서 찾을 수 있습니다.

버전 3.14에서 변경: Support for optionally dropping grouping parentheses when using multiple exception types. See **PEP 758**.

8.4.1 `except` clause

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the *try* clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the `except` clauses in turn until one is found that matches the exception. An expression-less `except` clause, if present, must be last; it matches any exception.

For an `except` clause with an expression, the expression must evaluate to an exception type or a tuple of exception types. Parentheses can be dropped if multiple exception types are provided and the `as` clause is not used. The raised exception matches an `except` clause whose expression evaluates to the class or a *non-virtual base class* of the exception object, or to a tuple that contains such a class.

If no `except` clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.[1]

If the evaluation of an expression in the header of an `except` clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire *try* statement raised the exception).

When a matching `except` clause is found, the exception is assigned to the target specified after the `as` keyword in that `except` clause, if present, and the `except` clause' s suite is executed. All `except` clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire *try* statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the `try` clause of the inner handler, the outer handler will not handle the exception.)

When an exception has been assigned using `as target`, it is cleared at the end of the `except` clause. This is as if

```
except E as N:
    foo
```

가 이렇게 변환되는 것과 같습니다

[1] 다른 예외를 일으키는 *finally* 절이 있지 않은 한 예외는 호출 스택으로 퍼집니다. 그 새 예외는 예전의 것을 잃어버리게 만듭니다.

```
except E as N:
    try:
        foo
    finally:
        del N
```

This means the exception must be assigned to a different name to be able to refer to it after the `except` clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Before an `except` clause' s suite is executed, the exception is stored in the `sys` module, where it can be accessed from within the body of the `except` clause by calling `sys.exception()`. When leaving an exception handler, the exception stored in the `sys` module is reset to its previous value:

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2 `except*` clause

The `except*` clause(s) are used for handling `ExceptionGroup`s. The exception type for matching is interpreted as in the case of *except*, but in the case of exception groups we can have partial matches when the type matches some of the exceptions in the group. This means that multiple `except*` clauses can execute, each handling part of the exception group. Each clause executes at most once and handles an exception group of all matching exceptions. Each exception in the group is handled by at most one `except*` clause, the first that matches it.

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

Any remaining exceptions that were not handled by any `except*` clause are re-raised at the end, along with all

exceptions that were raised from within the `except*` clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

If the raised exception is not an exception group and its type matches one of the `except*` clauses, it is caught and wrapped by an exception group with an empty message string.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

An `except*` clause must have a matching expression; it cannot be `except*:`. Furthermore, this expression cannot contain exception group types, because that would have ambiguous semantics.

It is not possible to mix *except* and `except*` in the same *try*. *break*, *continue* and *return* cannot appear in an `except*` clause.

8.4.3 `else` clause

생략 가능한 `else` 절은 제어 흐름이 *try* 스위트를 빠져나가고, 예외가 발생하지 않았고, *return*, *continue* 또는 *break* 문이 실행되지 않으면 실행됩니다. `else` 절에서 발생하는 예외는 앞에 나오는 *except* 절에서 처리되지 않습니다.

8.4.4 `finally` clause

If `finally` is present, it specifies a 'cleanup' handler. The *try* clause is executed, including any *except* and *else* clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception it is re-raised at the end of the `finally` clause. If the `finally` clause raises another exception, the saved exception is set as the context of the new exception. If the `finally` clause executes a *return*, *break* or *continue* statement, the saved exception is discarded. For example, this function returns 42.

```
def f():
    try:
        1/0
    finally:
        return 42
```

The exception information is not available to the program during execution of the `finally` clause.

When a *return*, *break* or *continue* statement is executed in the *try* suite of a `try`…`finally` statement, the `finally` clause is also executed 'on the way out.'

The return value of a function is determined by the last *return* statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed. The following function returns 'finally'.

```
def foo():
    try:
        return 'try'
    finally:
        return 'finally'
```

버전 3.8에서 변경: Prior to Python 3.8, a *continue* statement was illegal in the `finally` clause due to a problem with the implementation.

버전 3.14에서 변경: The compiler emits a `SyntaxWarning` when a *return*, *break* or *continue* appears in a `finally` block (see **PEP 765**).

8.5 with 문

with 문은 블록의 실행을 컨텍스트 관리자 (*with* 문 컨텍스트 관리자 섹션을 보세요) 가 정의한 메서드들로 감싸는 데 사용됩니다. 이것은 흔한 *try*…*except*…*finally* 사용 패턴을 편리하게 재사용할 수 있도록 캡슐화할 수 있도록 합니다.

```
with_stmt:          "with" ( "(" with_stmt_contents ","? ")" | with_stmt_contents ) ":" suite
with_stmt_contents: with_item ("," with_item)*
with_item:          expression ["as" target]
```

하나의 "item" 을 사용하는 *with* 문의 실행은 다음과 같이 진행됩니다:

1. The context expression (the expression given in the *with_item*) is evaluated to obtain a context manager.
2. The context manager's *__enter__()* is loaded for later use.
3. The context manager's *__exit__()* is loaded for later use.
4. The context manager's *__enter__()* method is invoked.
5. If a target was included in the *with* statement, the return value from *__enter__()* is assigned to it.

   > **참고**
   >
   > The *with* statement guarantees that if the *__enter__()* method returns without an error, then *__exit__()* will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 7 below.

6. 스위트가 실행됩니다.
7. The context manager's *__exit__()* method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to *__exit__()*. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the *__exit__()* method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the *with* statement.

   If the suite was exited for any reason other than an exception, the return value from *__exit__()* is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

다음과 같은 코드는:

```
with EXPRESSION as TARGET:
    SUITE
```

의미상으로 다음과 동등합니다:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

하나 보다 많은 항목을 주면, 컨텍스트 관리자는 *with* 문이 중첩된 것처럼 진행합니다:

```
with A() as a, B() as b:
    SUITE
```

의미상으로 다음과 동등합니다:

```
with A() as a:
    with B() as b:
        SUITE
```

You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses. For example:

```
with (
    A() as a,
    B() as b,
):
    SUITE
```

버전 3.1에서 변경: 다중 컨텍스트 표현식의 지원

버전 3.10에서 변경: Support for using grouping parentheses to break the statement in multiple lines.

더 보기

PEP 343 - "with" 문
파이썬 *with* 문의 규격, 배경, 예.

8.6 The `match` statement

Added in version 3.10.

The match statement is used for pattern matching. Syntax:

```
match_stmt:   'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr: star_named_expression "," star_named_expressions?
            | named_expression
case_block:   'case' patterns [guard] ":" block
```

참고

This section uses single quotes to denote *soft keywords*.

Pattern matching takes a pattern as input (following `case`) and a subject value (following `match`). The pattern (which may contain subpatterns) is matched against the subject value. The outcomes are:

- A match success or failure (also termed a pattern success or failure).
- Possible binding of matched values to a name. The prerequisites for this are further discussed below.

The `match` and `case` keywords are *soft keywords*.

더 보기

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Structural Pattern Matching: Tutorial

8.6.1 Overview

Here' s an overview of the logical flow of a match statement:

1. The subject expression `subject_expr` is evaluated and a resulting subject value obtained. If the subject expression contains a comma, a tuple is constructed using the standard rules.
2. Each pattern in a `case_block` is attempted to match with the subject value. The specific rules for success or failure are described below. The match attempt can also bind some or all of the standalone names within the pattern. The precise pattern binding rules vary per pattern type and are specified below. **Name bindings made during a successful pattern match outlive the executed block and can be used after the match statement**.

   참고

   During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match. The exact behavior is dependent on implementation and may vary. This is an intentional decision made to allow different implementations to add optimizations.

3. If the pattern succeeds, the corresponding guard (if present) is evaluated. In this case all name bindings are guaranteed to have happened.
   - If the guard evaluates as true or is missing, the `block` inside `case_block` is executed.
   - Otherwise, the next `case_block` is attempted as described above.
   - If there are no further case blocks, the match statement is completed.

참고

Users should generally never rely on a pattern being evaluated. Depending on implementation, the interpreter may cache values or use other optimizations which skip repeated evaluations.

A sample match statement:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Successful match, but guard fails
...         print('Case 2')
...     case (100, y):  # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _:  # Pattern not attempted
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

In this case, `if flag` is a guard. Read more about that in the next section.

8.6.2 Guards

```
guard: "if" named_expression
```

A `guard` (which is part of the `case`) must succeed for code inside the `case` block to execute. It takes the form: *if* followed by an expression.

The logical flow of a `case` block with a `guard` follows:

1. Check that the pattern in the `case` block succeeded. If the pattern failed, the `guard` is not evaluated and the next `case` block is checked.
2. If the pattern succeeded, evaluate the `guard`.
   - If the `guard` condition evaluates as true, the case block is selected.
   - If the `guard` condition evaluates as false, the case block is not selected.
   - If the `guard` raises an exception during evaluation, the exception bubbles up.

Guards are allowed to have side effects as they are expressions. Guard evaluation must proceed from the first to the last case block, one at a time, skipping case blocks whose pattern(s) don' t all succeed. (I.e., guard evaluation must happen in order.) Guard evaluation must stop once a case block is selected.

8.6.3 Irrefutable Case Blocks

An irrefutable case block is a match-all case block. A match statement may have at most one irrefutable case block, and it must be last.

A case block is considered irrefutable if it has no guard and its pattern is irrefutable. A pattern is considered irrefutable if we can prove from its syntax alone that it will always succeed. Only the following patterns are irrefutable:

- *AS Patterns* whose left-hand side is irrefutable
- *OR Patterns* containing at least one irrefutable pattern
- *Capture Patterns*
- *Wildcard Patterns*
- parenthesized irrefutable patterns

8.6.4 Patterns

> **참고**
>
> This section uses grammar notations beyond standard EBNF:
>
> - the notation `SEP.RULE+` is shorthand for `RULE (SEP RULE)*`
> - the notation `!RULE` is shorthand for a negative lookahead assertion

The top-level syntax for `patterns` is:

```
patterns:       open_sequence_pattern | pattern
pattern:        as_pattern | or_pattern
closed_pattern: | literal_pattern
                | capture_pattern
                | wildcard_pattern
                | value_pattern
                | group_pattern
                | sequence_pattern
                | mapping_pattern
                | class_pattern
```

The descriptions below will include a description "in simple terms" of what a pattern does for illustration purposes (credits to Raymond Hettinger for a document that inspired most of the descriptions). Note that these descriptions are purely for illustration purposes and **may not** reflect the underlying implementation. Furthermore, they do not cover all valid forms.

OR Patterns

An OR pattern is two or more patterns separated by vertical bars `|`. Syntax:

```
or_pattern: "|".closed_pattern+
```

Only the final subpattern may be *irrefutable*, and each subpattern must bind the same set of names to avoid ambiguity.

An OR pattern matches each of its subpatterns in turn to the subject value, until one succeeds. The OR pattern is then considered successful. Otherwise, if none of the subpatterns succeed, the OR pattern fails.

In simple terms, `P1 | P2 | ...` will try to match `P1`, if it fails it will try to match `P2`, succeeding immediately if any succeeds, failing otherwise.

AS Patterns

An AS pattern matches an OR pattern on the left of the `as` keyword against a subject. Syntax:

```
as_pattern: or_pattern "as" capture_pattern
```

If the OR pattern fails, the AS pattern fails. Otherwise, the AS pattern binds the subject to the name on the right of the as keyword and succeeds. `capture_pattern` cannot be a `_`.

In simple terms `P as NAME` will match with `P`, and on success it will set `NAME = <subject>`.

Literal Patterns

A literal pattern corresponds to most *literals* in Python. Syntax:

```
literal_pattern: signed_number
                 | signed_number "+" NUMBER
                 | signed_number "-" NUMBER
                 | strings
                 | "None"
                 | "True"
                 | "False"
signed_number:   ["-"] NUMBER
```

The rule `strings` and the token `NUMBER` are defined in the *standard Python grammar*. Triple-quoted strings are supported. Raw strings and byte strings are supported. *f-strings* are not supported.

The forms `signed_number '+' NUMBER` and `signed_number '-' NUMBER` are for expressing *complex numbers*; they require a real number on the left and an imaginary number on the right. E.g. `3 + 4j`.

In simple terms, `LITERAL` will succeed only if `<subject> == LITERAL`. For the singletons `None`, `True` and `False`, the `is` operator is used.

Capture Patterns

A capture pattern binds the subject value to a name. Syntax:

```
capture_pattern: !'_' NAME
```

A single underscore `_` is not a capture pattern (this is what `!'_'` expresses). It is instead treated as a `wildcard_pattern`.

In a given pattern, a given name can only be bound once. E.g. `case x, x: ...` is invalid while `case [x] | x: ...` is allowed.

Capture patterns always succeed. The binding follows scoping rules established by the assignment expression operator in **PEP 572**; the name becomes a local variable in the closest containing function scope unless there' s an applicable *global* or *nonlocal* statement.

In simple terms `NAME` will always succeed and it will set `NAME = <subject>`.

Wildcard Patterns

A wildcard pattern always succeeds (matches anything) and binds no name. Syntax:

```
wildcard_pattern: '_'
```

_ is a *soft keyword* within any pattern, but only within patterns. It is an identifier, as usual, even within `match` subject expressions, `guard`s, and `case` blocks.

In simple terms, _ will always succeed.

Value Patterns

A value pattern represents a named value in Python. Syntax:

```
value_pattern: attr
attr:          name_or_attr "." NAME
name_or_attr:  attr | NAME
```

The dotted name in the pattern is looked up using standard Python *name resolution rules*. The pattern succeeds if the value found compares equal to the subject value (using the == equality operator).

In simple terms `NAME1.NAME2` will succeed only if `<subject> == NAME1.NAME2`

> **참고**
>
> If the same value occurs multiple times in the same match statement, the interpreter may cache the first value found and reuse it rather than repeat the same lookup. This cache is strictly tied to a given execution of a given match statement.

Group Patterns

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
group_pattern: "(" pattern ")"
```

In simple terms `(P)` has the same effect as `P`.

Sequence Patterns

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
sequence_pattern:       "[" [maybe_sequence_pattern] "]"
                        | "(" [open_sequence_pattern] ")"
open_sequence_pattern:  maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern: ",".maybe_star_pattern+ ","?
maybe_star_pattern:     star_pattern | pattern
star_pattern:           "*" (capture_pattern | wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. `(...)` vs `[...]`).

> **참고**
>
> A single pattern enclosed in parentheses without a trailing comma (e.g. `(3 | 4)`) is a *group pattern*. While a single pattern enclosed in square brackets (e.g. `[3 | 4]`) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position. If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

1. If the subject value is not a sequence[2], the sequence pattern fails.
2. If the subject value is an instance of `str`, `bytes` or `bytearray` the sequence pattern fails.
3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

   If the sequence pattern is fixed-length:

   1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails
   2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching their corresponding item, the sequence pattern succeeds.

   Otherwise, if the sequence pattern is variable-length:

   1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern fails.
   2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.
   3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.
   4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.

   > **참고**
   >
   > The length of the subject sequence is obtained via `len()` (i.e. via the `__len__()` protocol). This length may be cached by the interpreter in a similar manner as *value patterns*.

In simple terms `[P1, P2, P3, … , P<N>]` matches only if all the following happens:

- check `<subject>` is a sequence
- `len(subject) == <N>`

[2] In pattern matching, a sequence is defined as one of the following:
- a class that inherits from `collections.abc.Sequence`
- a Python class that has been registered as `collections.abc.Sequence`
- a builtin class that has its (CPython) `Py_TPFLAGS_SEQUENCE` bit set
- a class that inherits from any of the above

The following standard library classes are sequences:
- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

> **참고**
>
> Subject values of type `str`, `bytes`, and `bytearray` do not match sequence patterns.

- `P1` matches `<subject>[0]` (note that this match can also bind names)
- `P2` matches `<subject>[1]` (note that this match can also bind names)
- ⋯ and so on for the corresponding pattern/element.

Mapping Patterns

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern:     "{" [items_pattern] "}"
items_pattern:       ",".key_value_pattern+ ","?
key_value_pattern:   (literal_pattern | value_pattern) ":" pattern
                     | double_star_pattern
double_star_pattern: "**" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a `SyntaxError`. Two keys that otherwise have the same value will raise a `ValueError` at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

1. If the subject value is not a mapping[3],the mapping pattern fails.
2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.
3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A `SyntaxError` is raised for duplicate literal values; or a `ValueError` for named keys of the same value.

> **참고**
>
> Key-value pairs are matched using the two-argument form of the mapping subject' s `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or *`__getitem__()`*.

In simple terms `{KEY1: P1, KEY2: P2, ... }` matches only if all the following happens:

- check `<subject>` is a mapping
- `KEY1 in <subject>`
- `P1` matches `<subject>[KEY1]`
- ⋯ and so on for the corresponding KEY/pattern pair.

Class Patterns

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

```
class_pattern:       name_or_attr "(" [pattern_arguments ","?] ")"
pattern_arguments:   positional_patterns ["," keyword_patterns]
                     | keyword_patterns
positional_patterns: ",".pattern+
keyword_patterns:    ",".keyword_pattern+
```

[3] In pattern matching, a mapping is defined as one of the following:
- a class that inherits from `collections.abc.Mapping`
- a Python class that has been registered as `collections.abc.Mapping`
- a builtin class that has its (CPython) `Py_TPFLAGS_MAPPING` bit set
- a class that inherits from any of the above

The standard library classes `dict` and `types.MappingProxyType` are mappings.

```
keyword_pattern:     NAME "=" pattern
```

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

1. If `name_or_attr` is not an instance of the builtin `type` , raise `TypeError`.
2. If the subject value is not an instance of `name_or_attr` (tested via `isinstance()`), the class pattern fails.
3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

   For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

   If only keyword patterns are present, they are processed as follows, one by one:

   I. The keyword is looked up as an attribute on the subject.

   - If this raises an exception other than `AttributeError`, the exception bubbles up.
   - If this raises `AttributeError`, the class pattern has failed.
   - Else, the subpattern associated with the keyword pattern is matched against the subject' s attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.

   II. If all keyword patterns succeed, the class pattern succeeds.

   If any positional patterns are present, they are converted to keyword patterns using the *__match_args__* attribute on the class `name_or_attr` before matching:

   I. The equivalent of `getattr(cls, "__match_args__", ())` is called.

   - If this raises an exception, the exception bubbles up.
   - If the returned value is not a tuple, the conversion fails and `TypeError` is raised.
   - If there are more positional patterns than `len(cls.__match_args__)`, `TypeError` is raised.
   - Otherwise, positional pattern `i` is converted to a keyword pattern using `__match_args__[i]` as the keyword. `__match_args__[i]` must be a string; if not `TypeError` is raised.
   - If there are duplicate keywords, `TypeError` is raised.

   > **더 보기**
   >
   > *Customizing positional arguments in class pattern matching*

   II. Once all positional patterns have been converted to keyword patterns,
   the match proceeds as if there were only keyword patterns.

   For the following built-in types the handling of positional subpatterns is different:

   - `bool`
   - `bytearray`
   - `bytes`
   - `dict`
   - `float`
   - `frozenset`
   - `int`
   - `list`
   - `set`

- `str`
- `tuple`

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example `int(0|1)` matches the value `0`, but not the value `0.0`.

In simple terms `CLS(P1, attr=P2)` matches only if the following happens:

- `isinstance(<subject>, CLS)`
- convert `P1` to a keyword pattern using `CLS.__match_args__`
- For each keyword argument `attr=P2`:
  - `hasattr(<subject>, "attr")`
  - `P2` matches `<subject>.attr`
- ⋯ and so on for the corresponding keyword argument/pattern pair.

더 보기

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Structural Pattern Matching: Tutorial

8.7 함수 정의

함수 정의는 사용자 정의 함수 객체 (표준형 계층 섹션을 보세요) 를 정의합니다:

```
funcdef:                    [decorators] "def" funcname [type_params] "(" [parameter_list] ")"
                            ["->" expression] ":" suite
decorators:                 decorator+
decorator:                  "@" assignment_expression NEWLINE
parameter_list:             defparameter ("," defparameter)* "," "/" ["," [parameter_list_no_poso
                              | parameter_list_no_posonly
parameter_list_no_posonly:  defparameter ("," defparameter)* ["," [parameter_list_starargs]]
                            | parameter_list_starargs
parameter_list_starargs:    "*" [star_parameter] ("," defparameter)* ["," [parameter_star_kwargs]
                            | "*" ("," defparameter)+ ["," [parameter_star_kwargs]]
                            | parameter_star_kwargs
parameter_star_kwargs:      "**" parameter [","]
parameter:                  identifier [":" expression]
star_parameter:             identifier [":" ["*"] expression]
defparameter:               parameter ["=" expression]
funcname:                   identifier
```

함수 정의는 실행할 수 있는 문장입니다. 실행하면 현재 지역 이름 공간의 함수 이름을 함수 객체 (함수의 실행 가능한 코드를 둘러싼 래퍼(wrapper)). 이 함수 객체는 현재의 이름 공간에 대한 참조를 포함하는데, 함수가 호출될 때 전역 이름 공간으로 사용됩니다.

함수 정의는 함수의 바디를 실행하지 않습니다. 함수가 호출될 때 실행됩니다.[4]

함수 정의는 하나나 그 이상의 데코레이터 표현식으로 감싸질 수 있습니다. 데코레이터 표현식은 함수가 정의될 때, 함수 정의를 포함하는 스코프에서 값을 구합니다. 그 결과는 콜러블이어야 하는데, 함수 객체만을 인자로 사용해서 호출됩니다. 반환 값이 함수 객체 대신 함수의 이름에 연결됩니다. 여러 개의 데코레이터는 중첩되는 방식으로 적용됩니다. 예를 들어, 다음과 같은 코드

[4] A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's *docstring*.

```
@f1(arg)
@f2
def func(): pass
```

는 대략 다음과 동등합니다

```
def func(): pass
func = f1(arg)(f2(func))
```

원래의 함수가 임시로 이름 `func` 에 연결되지 않는다는 점만 다릅니다.

버전 3.9에서 변경: Functions may be decorated with any valid *assignment_expression*. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's *__type_params__* attribute. See *Generic functions* for more.

버전 3.12에서 변경: Type parameter lists are new in Python 3.12.

하나나 그 이상의 매개변수 들이 *parameter* = *expression* 형태를 가질 때, 함수가 "기본 매개변수 값"을 갖는다고 말합니다. 기본값이 있는 매개변수의 경우, 호출할 때 대응하는 인자 를 생략할 수 있고, 그럴 때 매개변수의 기본값이 적용됩니다. 만약 매개변수가 기본값을 가지면, "`*`" 까지 그 뒤를 따르는 모든 매개변수도 기본값을 가져야 합니다 — 이것은 문법 규칙에서 표현되지 않는 문법적 제약입니다.

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same "pre-computed" value is used for each call. This is especially important to understand when a default parameter value is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default parameter value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section 호출. A function call always assigns values to all parameters mentioned in the parameter list, either from positional arguments, from keyword arguments, or from default values. If the form "`*identifier`" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "`**identifier`" is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after "`*`" or "`*identifier`" are keyword-only parameters and may only be passed by keyword arguments. Parameters before "`/`" are positional-only parameters and may only be passed by positional arguments.

버전 3.8에서 변경: The `/` function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** for details.

Parameters may have an *annotation* of the form "`: expression`" following the parameter name. Any parameter may have an annotation, even those of the form `*identifier` or `**identifier`. (As a special case, parameters of the form `*identifier` may have an annotation "`: *expression`".) Functions may have "return" annotation of the form "`-> expression`" after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. See *Annotations* for more information on annotations.

버전 3.11에서 변경: Parameters of the form "`*identifier`" may have an annotation "`: *expression`". See **PEP 646**.

표현식에서 즉시 사용하기 위해, 이름 없는 함수(이름에 연결되지 않은 함수)를 만드는 것도 가능합니다. 이것은 람다 표현식을 사용하는데, 람다(*Lambdas*) 섹션에서 설명합니다. 람다 표현식은 단순화된 함수 정의를 위한 줄임 표현에 지나지 않는다는 것에 주의하세요; "*def*" 문장에서 정의된 함수는 람다 표현

식으로 정의된 함수처럼 전달되거나 다른 이름에 대입될 수 있습니다. 여러 개의 문장을 실행하는 것과 어노테이션을 허락하기 때문에, "def" 형태가 사실 더 강력합니다.

프로그래머 유의 사항: 함수는 퍼스트 클래스(first-class) 객체다. 함수 정의 안에서 실행되는 "def" 문은 돌려주거나 전달할 수 있는 지역 함수를 정의합니다. 중첩된 함수에서 사용되는 자유 변수들은 그 def 를 포함하는 함수의 지역 변수들을 액세스할 수 있습니다. 더 자세한 내용은 이름과 연결(*binding*) 섹션을 보세요.

더 보기

PEP 3107 - 함수 어노테이션
함수 어노테이션의 최초 규격.

PEP 484 - 형 힌트
어노테이션에 대한 표준 의미 정의: 형 힌트.

PEP 526 - 변수 어노테이션 문법
Ability to type hint variable declarations, including class variables and instance variables.

PEP 563 - 어노테이션의 지연된 평가
즉시 평가하는 대신 실행시간에 어노테이션을 문자열 형식으로 보존하여 어노테이션 내에서의 전방 참조를 지원합니다.

PEP 318 - Decorators for Functions and Methods
Function and method decorators were introduced. Class decorators were introduced in **PEP 3129**.

8.8 클래스 정의

클래스 정의는 클래스 객체(표준형 계층 섹션을 보세요)를 정의합니다:

```
classdef:    [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance: "(" [argument_list] ")"
classname:   identifier
```

클래스 정의는 실행 가능한 문장입니다. 계승(inheritance) 목록은 보통 베이스 클래스들의 목록을 제공하는데 (더 고급 사용에 대해서는 메타 클래스 를 보세요), 목록의 각 항목은 값을 구할 때 서브 클래싱을 허락하는 클래스 객체가 되어야 합니다. 계승 목록이 없는 클래스는, 기본적으로, 베이스 클래스 `object` 를 계승합니다; 그래서

```
class Foo:
    pass
```

는 다음과 동등합니다

```
class Foo(object):
    pass
```

클래스의 스위트는 새로 만들어진 지역 이름 공간과 원래의 전역 이름 공간을 사용하는 새 실행 프레임(이름과 연결(*binding*) 을 보세요)에서 실행됩니다. (보통, 스위트는 대부분 함수 정의들을 포함합니다.) 클래스의 스위트가 실행을 마치면, 실행 프레임은 파기하지만, 그것의 지역 이름 공간은 보존합니다.[5] 그런 다음, 계승 목록을 베이스 클래스들로, 보존된 지역 이름 공간을 어트리뷰트 딕셔너리로 사용해서 새 클래스 객체를 만듭니다. 클래스의 이름은 원래의 지역 이름 공간에서 이 클래스 객체와 연결됩니다.

The order in which attributes are defined in the class body is preserved in the new class' s *__dict__*. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

클래스 생성은 메타 클래스 를 사용해서 심하게 커스터마이즈할 수 있습니다.

클래스 역시 함수를 데코레이팅할 때처럼 데코레이트할 수 있습니다,

[5] A string literal appearing as the first statement in the class body is transformed into the namespace' s *__doc__* item and therefore the class' s *docstring*.

```
@f1(arg)
@f2
class Foo: pass
```

는 대략 다음과 동등합니다

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

데코레이터 표현식의 값을 구하는 규칙은 함수 데코레이터와 같습니다. 그런 다음 그 결과가 클래스 이름에 연결됩니다.

버전 3.9에서 변경: Classes may be decorated with any valid *assignment_expression*. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets immediately after the class' s name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class' s *__type_params__* attribute. See *Generic classes* for more.

버전 3.12에서 변경: Type parameter lists are new in Python 3.12.

프로그래머 유의 사항: 클래스 정의에서 정의되는 변수들은 클래스 어트리뷰트입니다; 이것들은 인스턴스 간에 공유됩니다. 인스턴스 어트리뷰트는 메서드에서 `self.name = value` 로 설정될 수 있습니다. 클래스와 인스턴스 어트리뷰트 모두 "`self.name`" 표기법으로 액세스할 수 있고, 이런 식으로 액세스할 때 인스턴스 어트리뷰트는 같은 이름의 클래스 어트리뷰트를 가립니다. 클래스 어트리뷰트는 인스턴스 어트리뷰트의 기본값으로 사용될 수 있지만, 가변 값을 사용하는 것은 예상하지 않은 결과를 줄 수 있습니다. 디스크립터 를 다른 구현 상세를 갖는 인스턴스 변수를 만드는데 사용할 수 있습니다.

더 보기

PEP 3115 - 파이썬 3000의 메타 클래스
메타 클래스 선언을 현재 문법으로 변경하고, 메타 클래스가 있는 클래스를 구성하는 방법의 의미를 변경하는 제안.

PEP 3129 - 클래스 데코레이터
클래스 데코레이터를 추가하는 제안. 함수와 메서드 데코레이터는 **PEP 318**에서 도입되었습니다.

8.9 코루틴

Added in version 3.5.

8.9.1 코루틴 함수 정의

```
async_funcdef: [decorators] "async" "def" funcname "(" [parameter_list] ")"
               ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see *coroutine*). `await` expressions, `async for` and `async with` can only be used in the body of a coroutine function.

`async def` 문법으로 정의된 함수는 항상 코루틴 함수인데, `await` 나 `async` 키워드를 포함하지 않는 경우도 그렇습니다.

코루틴 함수의 바디 안에서 `yield from` 표현식을 사용하는 것은 `SyntaxError` 입니다.

코루틴 함수의 예:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

버전 3.7에서 변경: `await` and `async` are now keywords; previously they were only treated as such inside the body of a coroutine function.

8.9.2 `async for` 문

```
async_for_stmt: "async" for_stmt
```

비동기 이터러블 은 비동기 이터레이터 를 직접 반환하는 `__aiter__` 메서드를 제공하고, 비동기 이터레이터는 자신의 `__anext__` 메서드에서 비동기 코드를 호출할 수 있습니다.

`async for` 문은 비동기 이터러블에 대한 편리한 이터레이션을 허락합니다.

다음과 같은 코드는:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

의미상으로 다음과 동등합니다:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

See also *`__aiter__()`* and *`__anext__()`* for details.

코루틴 함수의 바디 밖에서 `async for` 문을 사용하는 것은 `SyntaxError` 입니다.

8.9.3 `async with` 문

```
async_with_stmt: "async" with_stmt
```

비동기 컨텍스트 관리자 는 *enter* 와 *exit* 메서드에서 실행을 일시 중지할 수 있는 컨텍스트 관리자 입니다.

다음과 같은 코드는:

```
async with EXPRESSION as TARGET:
    SUITE
```

의미상으로 다음과 동등합니다:

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

See also *__aenter__()* and *__aexit__()* for details.

코루틴 함수의 바디 밖에서 `async with` 문을 사용하는 것은 `SyntaxError` 입니다.

더 보기

PEP 492 - async 와 await 문법을 사용하는 코루틴
코루틴을 파이썬에서 적절한 독립적인 개념으로 만들고, 문법 지원을 추가한 제안.

8.10 Type parameter lists

Added in version 3.12.

버전 3.13에서 변경: Support for default values was added (see **PEP 696**).

```
type_params:  "[" type_param ("," type_param)* "]"
type_param:   typevar | typevartuple | paramspec
typevar:      identifier (":" expression)? ("=" expression)?
typevartuple: "*" identifier ("=" expression)?
paramspec:    "**" identifier ("=" expression)?
```

Functions (including *coroutines*), *classes* and *type aliases* may contain a type parameter list:

```
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T](): pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an *annotation scope*) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.

Type parameters come in three kinds:

- `typing.TypeVar`, introduced by a plain name (e.g., `T`). Semantically, this represents a single type to a type checker.
- `typing.TypeVarTuple`, introduced by a name prefixed with a single asterisk (e.g., `*Ts`). Semantically, this stands for a tuple of any number of types.
- `typing.ParamSpec`, introduced by a name prefixed with two asterisks (e.g., `**P`). Semantically, this stands for the parameters of a callable.

`typing.TypeVar` declarations can define *bounds* and *constraints* with a colon (`:`) followed by an expression. A single expression after the colon indicates a bound (e.g. `T: int`). Semantically, this means that the `typing.TypeVar` can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. `T: (str, bytes)`). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For `typing.TypeVar`s declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes `__bound__` and `__constraints__`. To accomplish this, the bounds or constraints are evaluated in a separate *annotation scope*.

`typing.TypeVarTuple`s and `typing.ParamSpec`s cannot have bounds or constraints.

All three flavors of type parameters can also have a *default value*, which is used when the type parameter is not explicitly provided. This is added by appending a single equals sign (`=`) followed by an expression. Like the bounds and constraints of type variables, the default value is not evaluated when the object is created, but only when the type parameter' s `__default__` attribute is accessed. To this end, the default value is evaluated in a separate *annotation scope*. If no default value is specified for a type parameter, the `__default__` attribute is set to the special sentinel object `typing.NoDefault`.

The following example indicates the full set of allowed type parameter declarations:

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithDefault = int,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple = (int, float),
   **SimpleParamSpec = (str, bytearray),
](
   a: SimpleTypeVar,
   b: TypeVarWithDefault,
   c: TypeVarWithBound,
   d: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *e: SimpleTypeVarTuple,
): ...
```

8.10.1 Generic functions

Generic functions are declared as follows:

```
def func[T](arg: T): ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here `annotation-def` indicates an *annotation scope*, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the `typing` module, but creates an instance of `typing.TypeVar` directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function' s defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Except for the *lazy evaluation* of the `TypeVar` bound, this is equivalent to:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like `DEFAULT_OF_arg` are not actually bound at runtime.

8.10.2 Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again `annotation-def` (not a real keyword) indicates an *annotation scope*, and the name `TYPE_PARAMS_OF_Bag` is not actually bound at runtime.

Generic classes implicitly inherit from `typing.Generic`. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

This is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 Generic type aliases

The *type* statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the *lazy evaluation* of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,
↪))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, `annotation-def` (not a real keyword) indicates an *annotation scope*. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

8.11 Annotations

버전 3.14에서 변경: Annotations are now lazily evaluated by default.

Variables and function parameters may carry *annotations*, created by adding a colon after the name, followed by an expression:

```
x: annotation = 1
def f(param: annotation): ...
```

Functions may also carry a return annotation following an arrow:

```
def f() -> annotation: ...
```

Annotations are conventionally used for *type hints*, but this is not enforced by the language, and in general annotations may contain arbitrary expressions. The presence of annotations does not change the runtime semantics of the code, except if some mechanism is used that introspects and uses the annotations (such as `dataclasses` or `functools.singledispatch()`).

By default, annotations are lazily evaluated in a *annotation scope*. This means that they are not evaluated when the code containing the annotation is evaluated. Instead, the interpreter saves information that can be used to evaluate the annotation later if requested. The `annotationlib` module provides tools for evaluating annotations.

If the *future statement* `from __future__ import annotations` is present, all annotations are instead stored as strings:

```
>>> from __future__ import annotations
>>> def f(param: annotation): ...
>>> f.__annotations__
{'param': 'annotation'}
```

CHAPTER 9

최상위 요소들

파이썬 인터프리터는 여러 가지 출처로부터 입력을 얻을 수 있습니다: 표준 입력이나 프로그램 인자로 전달된 스크립트, 대화형으로 입력된 것, 모듈 소스 파일 등등. 이 장은 이 경우들에 사용되는 문법을 제공합니다.

9.1 완전한 파이썬 프로그램

언어 규격이 어떤 식으로 언어 인터프리터가 실행되는지를 미리 규정할 필요는 없지만, 완전한 파이썬 프로그램이라는 개념을 갖는 것은 쓸모가 있습니다. 완전한 파이썬 프로그램은 최소한으로 초기화된 환경에서 실행됩니다: 모든 내장과 표준 모듈이 제공되지만, `sys` (각종 시스템 서비스들)와 `builtins` (내장 함수들, 예외들, `None`)과 `__main__` 이외의 어느 것도 초기화되지 않았습니다. 마지막 것은 완전한 프로그램의 실행을 위한 지역과 전역 이름 공간을 제공하는 데 사용됩니다.

완전한 파이썬 프로그램의 문법은 다음 섹션에서 설명되는 파일 입력의 경우입니다.

인터프리터는 대화형으로 실행될 수도 있습니다; 이 경우, 완전한 프로그램을 읽어서 실행하지 않고, 한 번에 한 문장 (복합문도 가능하다) 씩 읽어서 실행합니다. 초기 환경은 완전한 프로그램과 같습니다; 각 문장은 `__main__` 의 이름 공간에서 실행됩니다.

완전한 프로그램은 세 가지 형태로 인터프리터에게 전달될 수 있습니다: `-c` *string* 명령행 옵션으로, 첫 번째 명령행 인자로 전달된 파일로, 표준 입력으로. 파일이나 표준입력이 tty 장치면, 인터프리터는 대화형 모드로 돌입합니다; 그렇지 않으면 그 파일을 완전한 프로그램으로 실행합니다.

9.2 파일 입력

비대화형 파일로부터 읽힌 모든 입력은 같은 형태를 취합니다:

```
file_input: (NEWLINE | statement)* ENDMARKER
```

이 문법은 다음과 같은 상황에서 사용됩니다:

- (파일이나 문자열로부터 온) 완전한 파이썬 프로그램을 파싱할 때;
- 모듈을 파싱할 때;
- `exec()` 함수로 전달된 문자열을 파싱할 때;

9.3 대화형 입력

대화형 모드에서의 입력은 다음과 같은 문법 규칙을 사용합니다:

```
interactive_input: [stmt_list] NEWLINE | compound_stmt NEWLINE | ENDMARKER
```

(최상위) 복합문은 대화형 모드에서 빈 줄을 붙여줘야 함에 유념해야 합니다; 파서가 입력의 끝을 감지하는 데 필요합니다.

9.4 표현식 입력

표현식 입력을 위해 `eval()` 이 사용됩니다. 앞에 오는 공백을 무시합니다. `eval()` 의 문자열 인자는 다음과 같은 형식을 취해야 합니다:

```
eval_input: expression_list NEWLINE* ENDMARKER
```

CHAPTER 10

전체 문법 규격

이것은 CPython 구문 분석기를 생성하는 데 사용되는 문법에서 직접 파생된, 전체 파이썬 문법 규칙입니다 (Grammar/python.gram을 참조하십시오). 이 버전은 코드 생성과 에러 복구와 관련된 세부 정보를 생략합니다.

The notation is a mixture of EBNF and PEG. In particular, `&` followed by a symbol, token or parenthesized group indicates a positive lookahead (i.e., is required to match but not consumed), while `!` indicates a negative lookahead (i.e., is required *not* to match). We use the `|` separator to mean PEG' s "ordered choice" (written as `/` in traditional PEG grammars). See **PEP 617** for more details on the grammar' s syntax.

```
# PEG grammar for Python



# ========================= START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to the
#   rule:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
#   Python.
# e1 e2
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#       rule_name[return_type]:
#            | first_alt
#            | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated parse tree
#   does not include the separator. This is otherwise identical to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
# &&e
#   Eager parse e. The parser will not backtrack and will immediately
#   fail with SyntaxError if e cannot be parsed.
#

# STARTING RULES
# ==============

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER

# GENERAL STATEMENTS
# ==================

statements: statement+

statement:
    | compound_stmt
    | simple_stmts

single_compound_stmt:
    | compound_stmt

statement_newline:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | single_compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | pass_stmt
    | del_stmt
    | yield_stmt
    | assert_stmt
    | break_stmt
    | continue_stmt
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =================

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
         | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ annotated_rhs !'=' [TYPE_COMMENT]
    | single_target augassign ~ annotated_rhs

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

pass_stmt:
    | 'pass'

break_stmt:
    | 'break'

continue_stmt:
    | 'continue'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+

del_stmt:
    | 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
    | import_name
    | import_from

# Import statements
# -----------------

import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----------------

class_def:
    | decorators class_def_raw
    | class_def_raw

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ] ':' block

# Function definitions
# --------------------

function_def:
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_
↪comment] block
    | 'async' 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':'␣
↪[func_type_comment] block

# Function parameters
# -------------------

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '**' param_no_default

# One parameter.  This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression  | invalid_default

# If statement
# ------------

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# ---------------

while_stmt:
    | 'while' named_expression ':' block [else_block]

# For statement
# -------------

for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_
↪block]
    | 'async' 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block␣
↪[else_block]

# With statement
# --------------

with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' [TYPE_COMMENT] block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | 'async' 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'async' 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

# Try statement
# -------------

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
    | 'try' ':' block except_star_block+ [else_block] [finally_block]


# Except statement
# ----------------

except_block:
    | 'except' expressions ':' block
    | 'except' expression 'as' NAME  ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expressions ':' block
    | 'except' '*' expression 'as' NAME ':' block
finally_block:
    | 'finally' ':' block
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# Match statement
# ---------------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | '*' pattern_capture_target
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ','? '}'
    | '{' items_pattern ',' double_star_pattern ','? '}'
    | '{' items_pattern ','? '}'

items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ','? ')'
    | name_or_attr '(' keyword_patterns ','? ')'
    | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ')'

positional_patterns:
    | ','.pattern+

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# ---------------

type_alias:
    | "type" NAME [type_params] '=' expression

# Type parameter declaration
# --------------------------

type_params:
    | invalid_type_params
    | '[' type_param_seq ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound] [type_param_default]
    | '*' NAME [type_param_starred_default]
    | '**' NAME [type_param_default]

type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# EXPRESSIONS
# -----------

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']

star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# --------------------

comparison:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -----------------

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# --------------------

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power

power:
    | await_primary '**' factor
    | await_primary

# Primary elements
# ----------------

# Primary elements are things like "obj.something.something", "obj[something]",
↪"obj(something)", "obj" ...

await_primary:
    | 'await' primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# Lambda functions
# ----------------

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default*␣
↪[lambda_star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc

lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'

lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds

lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
# ========

fstring_middle:
    | fstring_replacement_field
    | FSTRING_MIDDLE
fstring_replacement_field:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | '{' annotated_rhs '='? [fstring_conversion] [fstring_full_format_spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FSTRING_MIDDLE
    | fstring_replacement_field
fstring:
    | FSTRING_START fstring_middle* FSTRING_END

string: STRING
strings: (fstring|string)+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',']

double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# ---------------------------

for_if_clauses:
    | for_if_clause+

for_if_clause:
    | 'async' 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=') for_if_clauses ')'

dictcomp:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =======================

arguments:
    | args [','] &')'

args:
    | ','.(starred_expression | ( assignment_expression | expression !':=') !'=')+␣
↪[',' kwargs ]
    | kwargs

kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression

# ASSIGNMENT TARGETS
# ==================

# Generic targets
# ---------------

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']

star_targets_list_seq: ','.star_target+ [',']

star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# --------------------------

del_targets: ','.del_target+ [',']

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be followed by indented block
    | TYPE_COMMENT
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# ========================= END OF THE GRAMMAR ===========================


# ========================= START OF INVALID RULES ========================
```

APPENDIX A

용어집

>>>
: 대화형 셸의 기본 파이썬 프롬프트. 인터프리터에서 대화형으로 실행될 수 있는 코드 예에서 자주 볼 수 있습니다.

...
: 다음과 같은 것들을 가리킬 수 있습니다:

  - 들여쓰기 된 코드 블록의 코드를 입력할 때, 쌍을 이루는 구분자 (괄호, 대괄호, 중괄호) 안에 코드를 입력할 때, 데코레이터 지정 후의 대화형 셸의 기본 파이썬 프롬프트.
  - `Ellipsis` 내장 상수.

abstract base class (추상 베이스 클래스)
: 추상 베이스 클래스는 `hasattr()` 같은 다른 테크닉들이 불편하거나 미묘하게 잘못된 (예를 들어, 매직 메서드) 경우, 인터페이스를 정의하는 방법을 제공함으로써 덕 타이핑 을 보완합니다. ABC는 가상 서브 클래스를 도입하는데, 클래스를 계승하지 않으면서도 `isinstance()` 와 `issubclass()` 에 의해 감지될 수 있는 클래스들입니다; `abc` 모듈 설명서를 보세요. 파이썬에는 많은 내장 ABC 들이 따라오는데 다음과 같은 것들이 있습니다: 자료 구조 (`collections.abc` 모듈에서), 숫자 (`numbers` 모듈에서), 스트림 (`io` 모듈에서), 임포트 파인더와 로더 (`importlib.abc` 모듈에서). `abc` 모듈을 사용해서 자신만의 ABC를 만들 수도 있습니다.

annotate function
: A function that can be called to retrieve the *annotations* of an object. This function is accessible as the *`__annotate__`* attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

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

  Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

  See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

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

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

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

- 위치 인자 (*positional argument*): 키워드 인자가 아닌 인자. 위치 인자들은 인자 목록의 처음에 나오거나 이터러블 의 앞에 `*` 를 붙여 전달할 수 있습니다. 예를 들어, 다음과 같은 호출에서 `3` 과 `5` 는 모두 위치 인자입니다.

```
complex(3, 5)
complex(*(3, 5))
```

인자는 함수 바디의 이름 붙은 지역 변수에 대입됩니다. 이 대입에 적용되는 규칙들에 대해서는 호출 절을 보세요. 문법적으로, 어떤 표현식이건 인자로 사용될 수 있습니다; 구해진 값이 지역 변수에 대입됩니다.

용어집의 매개변수 항목과 FAQ 질문 인자와 매개변수의 차이 와 **PEP 362**도 보세요.

asynchronous context manager (비동기 컨텍스트 관리자)
`__aenter__()` 와 *`__aexit__()`* 메서드를 정의함으로써 *`async with`* 문에서 보이는 환경을 제어하는 객체. **PEP 492**로 도입되었습니다.

asynchronous generator (비동기 제너레이터)
비동기 제너레이터 이터레이터 를 돌려주는 함수. *`async def`* 로 정의되는 코루틴 함수처럼 보이는데, *`async for`* 루프가 사용할 수 있는 일련의 값들을 만드는 *`yield`* 표현식을 포함한다는 점이 다릅니다.

보통 비동기 제너레이터 함수를 가리키지만, 어떤 문맥에서는 비동기 제너레이터 이터레이터 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

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

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

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

각 *`yield`*는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 이터레이터 가 *`__anext__()`* 가 돌려주는 또 하나의 어웨이터블로 재개되면, 떠난 곳으로 복귀합니다. **PEP 492**와 **PEP 525**를 보세요.

asynchronous iterable (비동기 이터러블)
`async for` 문에서 사용될 수 있는 객체. *`__aiter__()`* 메서드는 비동기 이터레이터 를 돌려줘야 합니다. **PEP 492** 로 도입되었습니다.

asynchronous iterator (비동기 이터레이터)
`__aiter__()` 와 *`__anext__()`* 메서드를 구현하는 객체. *`__anext__()`* 는 어웨이터블 객체를 돌려줘야 합니다. *`async for`*는 `StopAsyncIteration` 예외가 발생할 때까지 비동기 이터레이터의 *`__anext__()`* 메서드가 돌려주는 어웨이터블을 풉니다. **PEP 492**로 도입되었습니다.

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds

of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

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

It is possible to give an object an attribute whose name is not an identifier as defined by 식별자와 키워드, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

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

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

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

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

borrowed reference (빌린 참조)
In Python' s C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object (바이트열류 객체)
bufferobjects 를 지원하고 C-연속 버퍼를 익스포트 할 수 있습니다. 여러 공통 `memoryview` 객체들은 물론이고 `bytes`, `bytearray`, `array.array` 객체들을 포함합니다. 바이트열류 객체들은 바이너리 데이터를 다루는 여러 가지 연산들에 사용될 수 있습니다; 압축, 바이너리 파일로 저장, 소켓을 통한 전송 같은 것들이 있습니다.

어떤 연산들은 바이너리 데이터가 가변적일 필요가 있습니다. 이런 경우에 설명서는 종종 "읽고-쓰기 바이트열류 객체"라고 표현합니다. 가변 버퍼 객체의 예로는 `bytearray` 와 `bytearray` 의 `memoryview` 가 있습니다. 다른 연산들은 바이너리 데이터가 불변 객체 ("읽기 전용 바이트열류 객체")에 저장되도록 요구합니다; 이런 것들의 예로는 `bytes`와 `bytes` 객체의 `memoryview` 가 있습니다.

bytecode (바이트 코드)
파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현입니다. 바이트 코드는 `.pyc` 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만듭니다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있습니다). 이 "중간 언어" 는 각 바이트 코드에 대응하는 기계를 실행하는 가상 기계 에서 실행된다고 말합니다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 합니다.

바이트 코드 명령어들의 목록은 dis 모듈 설명서에 나옵니다.

callable (콜러블)
A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the *__call__()* method is also a callable.

callback (콜백)
인자로 전달되는 미래의 어느 시점에서 실행될 서브 루틴 함수.

class (클래스)
사용자 정의 객체들을 만들기 위한 주형. 클래스 정의는 보통 클래스의 인스턴스를 대상으로 연산하는 메서드 정의들을 포함합니다.

class variable (클래스 변수)
클래스에서 정의되고 클래스 수준 (즉, 클래스의 인스턴스에서가 아니라) 에서만 수정되는 변수.

closure variable (클로저 변수)
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number (복소수)
익숙한 실수 시스템의 확장인데, 모든 숫자가 실수부와 허수부의 합으로 표현됩니다. 허수부는 실수에 허수 단위(`-1`의 제곱근)를 곱한 것인데, 종종 수학에서는 `i`로, 공학에서는 `j`로 표기합니다. 파이썬은 후자의 표기법을 쓰는 복소수를 기본 지원합니다; 허수부는 `j` 접미사를 붙여서 표기합니다, 예를 들어, `3+1j`. `math` 모듈의 복소수 버전이 필요하면, `cmath`를 사용합니다. 복소수의 활용은 꽤 수준 높은 수학적 기능입니다. 필요하다고 느끼지 못한다면, 거의 확실히 무시해도 좋습니다.

context (컨텍스트)
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol (컨텍스트 관리 프로토콜)
The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

context manager (컨텍스트 관리자)
컨텍스트 관리 프로토콜을 구현하고 `with` 문에서 보이는 환경을 제어하는 객체. **PEP 343**을 참조하십시오.

context variable (컨텍스트 변수)
A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

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

coroutine (코루틴)
코루틴은 서브루틴의 더 일반화된 형태입니다. 서브루틴은 한 지점에서 진입하고 다른 지점에서 탈출합니다. 코루틴은 여러 다른 지점에서 진입하고, 탈출하고, 재개할 수 있습니다. 이것들은 *async def* 문으로 구현할 수 있습니다. **PEP 492**를 보세요.

coroutine function (코루틴 함수)
코루틴 객체를 돌려주는 함수. 코루틴 함수는 *async def* 문으로 정의될 수 있고, *await* 와 *async for*와 *async with* 키워드를 포함할 수 있습니다. 이것들은 **PEP 492** 에 의해 도입되었습니다.

CPython
파이썬 프로그래밍 언어의 규범적인 구현인데, python.org에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 "CPython" 이 사용됩니다.

current context (현재 컨텍스트)
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

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

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

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

@staticmethod
def f(arg):
    ...
```

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의 와 클래스 정의 의 설명서를 보면 됩니다.

descriptor (디스크립터)
메서드 *__get__()* 이나 *__set__()* 이나 *__delete__()*를 정의하는 객체. 클래스 어트리뷰트가 디스크립터일 때, 어트리뷰트 조회는 특별한 연결 작용을 일으킵니다. 보통, *a.b*를 읽거나, 쓰거나, 삭제하는데 사용할 때, *a*의 클래스 딕셔너리에서 *b* 라고 이름 붙여진 객체를 찾습니다. 하지만 *b*가 디스크립터면, 해당하는 디스크립터 메서드가 호출됩니다. 디스크립터를 이해하는 것은 파이썬에 대한 깊은 이해의 열쇠인데, 함수, 메서드, 프로퍼티, 클래스 메서드, 스태틱 메서드, 슈퍼 클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 디스크립터 구현하기나 디스크립터 사용법 안내서에 나옵니다.

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

dictionary comprehension (딕셔너리 컴프리헨션)
이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 딕셔너리를 반환하는 간결한 방법. `results = {n: n ** 2 for n in range(10)}`은 값 `n ** 2`에 매핑된 키 `n`을 포함하는 딕셔너리를 생성합니다. 리스트, 집합, 딕셔너리의 디스플레이(*display*)을 참조하십시오.

dictionary view (딕셔너리 뷰)
`dict.keys()`, `dict.values()`, `dict.items()` 메서드가 돌려주는 객체들을 딕셔너리 뷰라고 부릅니다. 이것들은 딕셔너리 항목들에 대한 동적인 뷰를 제공하는데, 딕셔너리가 변경될 때, 뷰가 이 변화를 반영한다는 뜻입니다. 딕셔너리 뷰를 완전한 리스트로 바꾸려면 `list(dictview)` 를 사용하면 됩니다. dict-views를 보세요.

docstring (독스트링)
클래스, 함수, 모듈에서 첫 번째 표현식으로 나타나는 문자열 리터럴. 스위트가 실행될 때는 무시되

지만, 컴파일러에 의해 인지되어 둘러싼 클래스, 함수, 모듈의 `__doc__` 어트리뷰트로 삽입됩니다. 인트로스펙션을 통해 사용할 수 있으므로, 객체의 설명서를 위한 규범적인 장소입니다.

duck-typing (덕 타이핑)
올바른 인터페이스를 가졌는지 판단하는데 객체의 형을 보지 않는 프로그래밍 스타일; 대신, 단순히 메서드나 어트리뷰트가 호출되거나 사용됩니다 ("오리처럼 보이고 오리처럼 꽥꽥댄다면, 그것은 오리다.") 특정한 형 대신에 인터페이스를 강조함으로써, 잘 설계된 코드는 다형적인 치환을 허락함으로써 유연성을 개선할 수 있습니다. 덕 타이핑은 `type()` 이나 `isinstance()` 을 사용한 검사를 피합니다. (하지만, 덕 타이핑이 추상 베이스 클래스 로 보완될 수 있음에 유의해야 합니다.) 대신에, `hasattr()` 검사나 *EAFP* 프로그래밍을 씁니다.

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

evaluate function
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the *type* statement.

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

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

f-string (f-문자열)
`'f'` 나 `'F'` 를 앞에 붙인 문자열 리터럴들을 흔히 "f-문자열" 이라고 부르는데, 포맷 문자열 리터럴 의 줄임말입니다. **PEP 498** 을 보세요.

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

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

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

filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기)
Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

로케일 인코딩 도 보세요.

finder (파인더)
임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

두 종류의 파인더가 있습니다: `sys.meta_path` 와 함께 사용하는 메타 경로 파인더 와 `sys.path_hooks` 과 함께 사용하는 경로 엔트리 파인더.

더 자세한 내용은 파인더(*finder*)와 로더(*loader*) 와 `importlib`를 참조하십시오.

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

free threading (자유 스레딩)
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable (자유 변수)
Formally, as defined in the *language execution model*, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function (함수)
호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 인자 가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. 매개변수 와 메서드 와 함수 정의 섹션도 보세요.

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

함수 어노테이션은 일반적으로 형 힌트 로 사용됩니다: 예를 들어, 이 함수는 두 개의 `int` 인자를 받아들일 것으로 기대되고, 동시에 `int` 반환 값을 줄 것으로 기대됩니다:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

함수 어노테이션 문법은 함수 정의 절에서 설명합니다.

이 기능을 설명하는 변수 어노테이션 과 **PEP 484**를 참조하세요. 또한 어노테이션에 대한 모범 사례는 annotations-howto를 참조하세요.

__future__
A *future statement*, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

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

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

generator (제너레이터)
제너레이터 이터레이터 를 돌려주는 함수. 일반 함수처럼 보이는데, 일련의 값들을 만드는 *yield* 표현식을 포함한다는 점이 다릅니다. 이 값들은 for-루프로 사용하거나 `next()` 함수로 한 번에 하나씩 꺼낼 수 있습니다.

보통 제너레이터 함수를 가리키지만, 어떤 문맥에서는 제너레이터 이터레이터 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

generator iterator (제너레이터 이터레이터)
제너레이터 함수가 만드는 객체.

각 *yield*는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 제너레이터 이터레이터 가 재개되면, 떠난 곳으로 복귀합니다 (호출마다 새로 시작하는 함수와 대비됩니다).

generator expression (제너레이터 표현식)
이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 `for` 절과 생략 가능한 `if` 절이 뒤에 붙는 일반 표현식 처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

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

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

싱글 디스패치 용어집 항목과 `functools.singledispatch()` 데코레이터와 **PEP 443**도 보세요.

generic type (제네릭 형)
매개 변수화 할 수 있는 형; 일반적으로 `list` 나 `dict`와 같은 컨테이너 클래스. 형 힌트와 어노테이션에 사용됩니다.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL
전역 인터프리터 록 을 보세요.

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

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

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

In prior versions of Python' s C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

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

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

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

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

IDLE
파이썬을 위한 통합 개발 및 학습 환경 (Integrated Development and Learning Environment). idle은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

immortal (불멸)
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via `PyUnstable_IsImmortal()` in the C API.

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

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

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

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

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

interpreted (인터프리티드)
바이트 코드 컴파일러의 존재 때문에 그 구분이 흐릿해지기는 하지만, 파이썬은 컴파일 언어가 아니라 인터프리터 언어입니다. 이것은 명시적으로 실행 파일을 만들지 않고도, 소스 파일을 직접 실행할 수 있다는 뜻입니다. 그 프로그램이 좀 더 천천히 실행되기는 하지만, 인터프리터 언어는 보통 컴파일 언어보다 짧은 개발/디버깅 주기를 갖습니다. 대화형 도 보세요.

interpreter shutdown (인터프리터 종료)
종료하라는 요청을 받을 때, 파이썬 인터프리터는 특별한 시기에 진입하는데, 모듈이나 여러 가지 중요한 내부 구조들과 같은 모든 할당된 자원들을 단계적으로 반납합니다. 또한, 가비지 수거기 를 여러 번 호출합니다. 사용자 정의 파괴자나 weakref 콜백에 있는 코드들의 실행을 시작시킬 수 있습니다. 종료 시기 동안 실행되는 코드는 다양한 예외들을 만날 수 있는데, 그것이 의존하는 자원들이 더 기능하지 않을 수 있기 때문입니다 (흔한 예는 라이브러리 모듈이나 경고 장치들입니다).

인터프리터 종료의 주된 원인은 실행되는 `__main__` 모듈이나 스크립트가 실행을 끝내는 것입니다.

iterable (이터러블)
멤버들을 한 번에 하나씩 돌려줄 수 있는 객체. 이터러블의 예로는 모든 (`list`, `str`, `tuple` 같은) 시퀀스 형들, `dict` 같은 몇몇 비 시퀀스 형들, 파일 객체들, *`__iter__()`* 나 시퀀스 개념을 구현하는 *`__getitem__()`* 메서드를 써서 정의한 모든 클래스의 객체들이 있습니다.

이터러블은 *`for`* 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 (`zip()`, `map()`, …) 에 사용될 수 있습니다. 이터러블 객체가 내장 함수 `iter()` 에 인자로 전달되면, 그 객체의 이터레이터를 돌려줍니다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효합니다. 이터러블을 사용할 때, 보통은 `iter()` 를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없습니다. *`for`* 문은 이것들을 여러분을 대신해서 자동으로 해주는데, 루프를 도는 동안 이터레이터를 잡아둘 이름 없는 변수를 만듭니다. 이터레이터, 시퀀스, 제너레이터 도 보세요.

iterator (이터레이터)
데이터의 스트림을 표현하는 객체. 이터레이터의 `__next__()` 메서드를 반복적으로 호출하면 (또는 내장 함수 `next()` 로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 `StopIteration` 예외를 일으킵니다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 `__next__()` 메서드 호출은 `StopIteration` 예외를 다시 일으키기만 합니다. 이터레이터는 이터레이터 객체 자신을 돌려주는 `__iter__()` 메서드를 가질 것이 요구되기 때문에, 이터레이터는

이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드입니다. (`list` 같은) 컨테이너 객체는 `iter()` 함수로 전달하거나 *for* 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

typeiter 에 더 자세한 내용이 있습니다.

CPython 구현 상세: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

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

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

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

keyword argument (키워드 인자)
인자 를 보세요.

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

LBYL
뛰기 전에 보라 (Look before you leap). 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 사전 조건들을 검사합니다. 이 스타일은 *EAFP* 접근법과 대비되고, 많은 *if* 문의 존재로 특징지어집니다.

다중 스레드 환경에서, LBYL 접근법은 "보기"와 "뛰기" 간에 경쟁 조건을 만들게 될 위험이 있습니다. 예를 들어, 코드 `if key in mapping: return mapping[key]` 는 검사 후에, 하지만 조회 전에, 다른 스레드가 *key*를 *mapping*에서 제거하면 실패할 수 있습니다. 이런 이슈는 록이나 EAFP 접근법을 사용함으로써 해결될 수 있습니다.

lexical analyzer (어휘 분석기)
Formal name for the *tokenizer*; see *token*.

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

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

loader (로더)
An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also:

- 파인더 (*finder*) 와 로더 (*loader*)
- `importlib.abc.Loader`
- **PEP 302**

locale encoding (로케일 인코딩)
On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method (매직 메서드)
특수 메서드 의 비공식적인 비슷한 말.

mapping (매핑)
임의의 키 조회를 지원하고 `collections.abc.Mapping` 이나 `collections.abc.MutableMapping` 추상 베이스 클래스 에 지정된 메서드들을 구현하는 컨테이너 객체. 예로는 `dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` 를 들 수 있습니다.

meta path finder (메타 경로 파인더)
`sys.meta_path` 의 검색이 돌려주는 파인더. 메타 경로 파인더는 경로 엔트리 파인더 와 관련되어 있기는 하지만 다릅니다.

메타 경로 파인더가 구현하는 메서드들에 대해서는 `importlib.abc.MetaPathFinder` 를 보면 됩니다.

metaclass (메타 클래스)
클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 딕셔너리, 베이스 클래스들의 목록을 만듭니다. 메타 클래스는 이 세 인자를 받아서 클래스를 만드는 책임을 집니다. 대부분의 객체 지향형 프로그래밍 언어들은 기본 구현을 제공합니다. 파이썬을 특별하게 만드는 것은 커스텀 메타 클래스를 만들 수 있다는 것입니다. 대부분 사용자에게는 이 도구가 전혀 필요 없지만, 필요가 생길 때, 메타 클래스는 강력하고 우아한 해법을 제공합니다. 어트리뷰트 액세스의 로깅(logging), 스레드 안전성의 추가, 객체 생성 추적, 싱글톤 구현과 많은 다른 작업에 사용됐습니다.

메타 클래스 에서 더 자세한 내용을 찾을 수 있습니다.

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

method resolution order (메서드 결정 순서)
메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서입니다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 python_2.3_mro를 보세요.

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

패키지 도 보세요.

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

Module specs 도 보세요.

MRO
메서드 결정 순서 를 보세요.

mutable (가변)
가변 객체는 값이 변할 수 있지만 `id()` 는 일정하게 유지합니다. 불변 도 보세요.

named tuple (네임드 튜플)
"named tuple(네임드 튜플)" 이라는 용어는 튜플에서 상속하고 이름 붙은 어트리뷰트를 사용하여 인덱스 할 수 있는 요소에 액세스 할 수 있는 모든 형이나 클래스에 적용됩니다. 형이나 클래스에는 다른 기능도 있을 수 있습니다.

time.localtime() 과 os.stat() 가 반환한 값을 포함하여, 여러 내장형이 네임드 튜플입니다. 또 다른 예는 sys.float_info 입니다:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

일부 네임드 튜플은 내장형(위의 예)입니다. 또는, tuple에서 상속하고 이름 붙은 필드를 정의하는 일반 클래스 정의로 네임드 튜플을 만들 수 있습니다. 이러한 클래스는 직접 작성하거나, typing.NamedTuple를 계승하거나 팩토리 함수 collections.namedtuple()로 만들 수 있습니다. 후자의 기법은 직접 작성하거나 내장 네임드 튜플에서는 찾을 수 없는 몇 가지 추가 메서드를 추가하기도 합니다.

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

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

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and 이름 공간 패키지.

모듈 도 보세요.

nested scope (중첩된 스코프)
둘러싼 정의에서 변수를 참조하는 능력. 예를 들어, 다른 함수 내부에서 정의된 함수는 바깥 함수에 있는 변수들을 참조할 수 있습니다. 중첩된 스코프는 기본적으로는 참조만 가능할 뿐, 대입은 되지 않는다는 것에 주의해야 합니다. 지역 변수들은 가장 내부의 스코프에서 읽고 씁니다. 마찬가지로, 전역 변수들은 전역 이름 공간에서 읽고 씁니다. *nonlocal* 은 바깥 스코프에 쓰는 것을 허락합니다.

new-style class (뉴스타일 클래스)
지금은 모든 클래스 객체에 사용되고 있는 클래스 버전의 예전 이름. 초기의 파이썬 버전에서는, 오직 뉴스타일 클래스만 *__slots__*, 디스크립터, 프라퍼티, *__getattribute__()*, 클래스 메서드, 스태틱 메서드와 같은 파이썬의 새롭고 다양한 기능들을 사용할 수 있었습니다.

object (객체)
상태 (어트리뷰트나 값) 를 갖고 동작 (메서드) 이 정의된 모든 데이터. 또한, 모든 뉴스타일 클래스의 최종적인 베이스 클래스입니다.

optimized scope (최적화된 스코프)
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package (패키지)
서브 모듈들이나, 재귀적으로 서브 패키지들을 포함할 수 있는 파이썬 모듈. 기술적으로, 패키지는 __path__ 어트리뷰트가 있는 파이썬 모듈입니다.

정규 패키지 와 이름 공간 패키지 도 보세요.

parameter (매개변수)
함수 (또는 메서드) 정의에서 함수가 받을 수 있는 인자 (또는 어떤 경우 인자들) 를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있습니다:

- 위치-키워드 (*positional-or-keyword*): 위치 인자 나 키워드 인자 로 전달될 수 있는 인자를 지정합니다. 이것이 기본 형태의 매개변수입니다, 예를 들어 다음에서 *foo* 와 *bar*:

```
def func(foo, bar=None): ...
```

- 위치-전용 (*positional-only*): 위치로만 제공될 수 있는 인자를 지정합니다. 위치 전용 매개변수는 함수 정의의 매개변수 목록에 / 문자를 포함하고 그 뒤에 정의할 수 있습니다, 예를 들어 다음에서 *posonly1* 과 *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- 키워드-전용 (*keyword-only*): 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 * 를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 *kw_only1* 와 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- 가변-위치 (*var-positional*): (다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정합니다. 이런 매개변수는 매개변수 이름에 * 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 다음에서 *args*:

```
def func(*args, **kwargs): ...
```

- 가변-키워드 (*var-keyword*): (다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정합니다. 이런 매개변수는 매개변수 이름에 ** 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 위의 예에서 *kwargs*.

매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있습니다.

인자 용어집 항목, 인자와 매개변수의 차이에 나오는 FAQ 질문, `inspect.Parameter` 클래스, 함수 정의 절, **PEP 362**도 보세요.

path entry (경로 엔트리)
경로 기반 파인더 가 임포트 할 모듈들을 찾기 위해 참고하는 임포트 경로 상의 하나의 장소.

path entry finder (경로 엔트리 파인더)
`sys.path_hooks` 에 있는 콜러블 (즉, 경로 엔트리 훅) 이 돌려주는 파인더 인데, 주어진 경로 엔트리 로 모듈을 찾는 방법을 알고 있습니다.

경로 엔트리 파인더들이 구현하는 메서드들은 `importlib.abc.PathEntryFinder` 에 나옵니다.

path entry hook (경로 엔트리 훅)
`sys.path_hooks` 리스트에 있는 콜러블인데, 특정 경로 엔트리 에서 모듈을 찾는 법을 알고 있다면 경로 엔트리 파인더 를 돌려줍니다.

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

path-like object (경로류 객체)
파일 시스템 경로를 나타내는 객체. 경로류 객체는 경로를 나타내는 `str` 나 `bytes` 객체이거나 `os.PathLike` 프로토콜을 구현하는 객체입니다. `os.PathLike` 프로토콜을 지원하는 객체는 `os.fspath()` 함수를 호출해서 `str` 나 `bytes` 파일 시스템 경로로 변환될 수 있습니다; 대신 `os.fsdecode()` 와 `os.fsencode()` 는 각각 `str` 나 `bytes` 결과를 보장하는데 사용될 수 있습니다. **PEP 519**로 도입되었습니다.

PEP
파이썬 개선 제안. PEP는 파이썬 커뮤니티에 정보를 제공하거나 파이썬 또는 그 프로세스 또는 환경에 대한 새로운 기능을 설명하는 설계 문서입니다. PEP는 제안된 기능에 대한 간결한 기술 사양 및 근거를 제공해야 합니다.

PEP는 주요 새로운 기능을 제안하고 문제에 대한 커뮤니티 입력을 수집하며 파이썬에 들어간 설계 결정을 문서로 만들기 위한 기본 메커니즘입니다. PEP 작성자는 커뮤니티 내에서 합의를 구축하고 반대 의견을 문서화 할 책임이 있습니다.

PEP 1 참조하세요.

portion (포션)
PEP 420 에서 정의한 것처럼, 이름 공간 패키지에 이바지하는 하나의 디렉터리에 들어있는 파일들의 집합 (zip 파일에 저장되는 것도 가능합니다).

positional argument (위치 인자)
인자 를 보세요.

provisional API (잠정 API)
잠정 API는 표준 라이브러리의 과거 호환성 보장으로부터 신중히 제외된 것입니다. 인터페이스의 큰 변화가 예상되지는 않지만, 잠정적이라고 표시되는 한, 코어 개발자들이 필요하다고 생각한다면 과거 호환성이 유지되지 않는 변경이 일어날 수 있습니다. 그런 변경은 불필요한 방식으로 일어나지는 않을 것입니다 — API를 포함하기 전에 놓친 중대하고 근본적인 결함이 발견된 경우에만 일어날 것입니다.

잠정 API에서조차도, 과거 호환성이 유지되지 않는 변경은 "최후의 수단"으로 여겨집니다 - 모든 식별된 문제들에 대해 과거 호환성을 유지하는 해법을 찾으려는 모든 시도가 선행됩니다.

이 절차는 표준 라이브러리가 오랜 시간 동안 잘못된 설계 오류에 발목 잡히지 않고 발전할 수 있도록 만듭니다. 더 자세한 내용은 **PEP 411**을 보면 됩니다.

provisional package (잠정 패키지)
잠정 *API* 를 보세요.

Python 3000 (파이썬 3000)
파이썬 3.x 배포 라인의 별명 (버전 3의 배포가 먼 미래의 이야기던 시절에 만들어진 이름이다.) 이것을 "Py3k" 로 줄여 쓰기도 합니다.

Pythonic (파이썬다운)
다른 언어들에서 일반적인 개념들을 사용해서 코드를 구현하는 대신, 파이썬 언어에서 가장 자주 사용되는 이디엄들을 가까이 따르는 아이디어나 코드 조각. 예를 들어, 파이썬에서 자주 쓰는 이디엄은 *for* 문을 사용해서 이터러블의 모든 요소로 루핑하는 것입니다. 다른 많은 언어에는 이런 종류의 구성물이 없으므로, 파이썬에 익숙하지 않은 사람들은 대신에 숫자 카운터를 사용하기도 합니다:

```
for i in range(len(food)):
    print(food[i])
```

더 깔끔한, 파이썬다운 방법은 이렇습니다:

```
for piece in food:
    print(piece)
```

qualified name (정규화된 이름)
모듈의 전역 스코프에서 모듈에 정의된 클래스, 함수, 메서드에 이르는 "경로"를 보여주는 점으로 구분된 이름. **PEP 3155** 에서 정의됩니다. 최상위 함수와 클래스의 경우에, 정규화된 이름은 객체의 이름과 같습니다:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

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

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

reference count (참조 횟수)
객체에 대한 참조의 개수. 객체의 참조 횟수가 0으로 떨어지면, 메모리가 반납됩니다. 일부 객체는 불멸이며 참조 횟수가 수정되지 않아서, 객체가 할당 해제되지 않습니다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소입니다. 프로그래머는 특정 객체의 참조 횟수를 돌려주는 `sys.getrefcount()` 함수를 호출할 수 있습니다.

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

이름 공간 패키지 도 보세요.

REPL
An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

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

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

`collections.abc.Sequence` 추상 베이스 클래스는 *`__getitem__()`* 과 *`__len__()`*을 넘어서 훨씬 풍부한 인터페이스를 정의하는데, `count()`, `index()`, *`__contains__()`*, *`__reversed__()`*를 추가합니다. 이 확장된 인터페이스를 구현한 형을 `register()`를 사용해서 명시적으로 등록할 수 있습니다. 시퀀스 메서드 일반에 대한 자세한 문서는, 공통 시퀀스 연산을 참조하세요.

set comprehension (집합 컴프리헨션)
이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 집합을 반환하는 간결한 방법. `results = {c for c in 'abracadabra' if c not in 'abc'}`는 문자열의 집합 `{'r', 'd'}`를 생성합니다. 리스트, 집합, 딕셔너리의 디스플레이(*display*)을 참조하십시오.

single dispatch (싱글 디스패치)
구현이 하나의 인자의 형에 기초해서 결정되는 제네릭 함수 디스패치의 한 형태.

slice (슬라이스)
보통 시퀀스 의 일부를 포함하는 객체. 슬라이스는 서브 스크립트 표기법을 사용해서 만듭니다. `variable_name[1:3:5]` 처럼, `[]` 안에서 여러 개의 숫자를 콜론으로 분리합니다. 대괄호 (서브 스크립트) 표기법은 내부적으로 `slice` 객체를 사용합니다.

soft deprecated (약하게 폐지된)
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

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

static type checker (정적 형 검사기)
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

strong reference (강한 참조)
In Python' s C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

빌린 참조 도 보세요.

text encoding (텍스트 인코딩)
A string in Python is a sequence of Unicode code points (in range `U+0000–U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization codecs, which are collectively referred to as "text encodings".

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

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

thread state
The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python' s C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See Thread State and the Global Interpreter Lock for more information.

token (토큰)
A small unit of source code, generated by the *lexical analyzer* (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python' s lexical analyzer. The `token` module contains information on the various types of tokens.

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

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

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

형 에일리어스는 형 힌트를 단순화하는 데 유용합니다. 예를 들면:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
Color = tuple[int, int, int]

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

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

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

형 힌트는 선택 사항이고 파이썬에서 강제되지는 않지만, 정적 형 검사기에 유용합니다. 또한 IDE의 코드 완성 및 리팩토링을 돕습니다.

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

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

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

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

변수 또는 클래스 어트리뷰트에 어노테이션을 달 때 대입은 선택 사항입니다:

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

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

```
count: int = 0
```

변수 어노테이션 문법은 섹션 어노테이트된 대입문(*Annotated assignment statements*) 에서 설명합니다.

이 기능을 설명하는 함수 어노테이션, **PEP 484** 및 **PEP 526**을 참조하세요. 또한 어노테이션 작업에 대한 모범 사례는 annotations-howto를 참조하세요.

virtual environment (가상 환경)
파이썬 사용자와 응용 프로그램이, 같은 시스템에서 실행되는 다른 파이썬 응용 프로그램들의 동작에 영향을 주지 않으면서, 파이썬 배포 패키지들을 설치하거나 업그레이드하는 것을 가능하게 하는, 협력적으로 격리된 실행 환경.

`venv` 도 보세요.

virtual machine (가상 기계)
소프트웨어만으로 정의된 컴퓨터. 파이썬의 가상 기계는 바이트 코드 컴파일러가 출력하는 바이트 코드를 실행합니다.

Zen of Python (파이썬 젠)
파이썬 디자인 원리와 철학들의 목록인데, 언어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 "`import this`" 를 입력하면 보입니다.

APPENDIX B

이 설명서에 관하여

파이썬 설명서는 reStructuredText 소스에서 만들어진 것으로, 원래 파이썬을 위해 제작되었고 이제는 독립 프로젝트로 유지 관리되는 설명서 생성기인 Sphinx 를 사용했습니다.

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

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 저자;
- reStructuredText와 Docutils 스위트를 만드는 Docutils 프로젝트.
- Fredrik Lundh, 그의 대안 파이썬 참조(Alternative Python Reference) 프로젝트에서 Sphinx가 많은 아이디어를 얻었습니다.

B.1 파이썬 설명서의 공헌자들

많은 사람이 파이썬 언어, 파이썬 표준 라이브러리 및 파이썬 설명서에 기여했습니다. 기여자의 부분적인 목록은 파이썬 소스 배포판의 Misc/ACKS 를 참조하십시오.

파이썬이 이런 멋진 설명서를 갖게 된 것은 파이썬 커뮤니티의 입력과 기여 때문입니다 – 감사합니다!

APPENDIX C

역사와 라이센스

C.1 소프트웨어의 역사

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python' s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

배포판	파생된 곳	해	소유자	GPL-compatible? (1)
0.9.0 ~ 1.2	n/a	1991-1995	CWI	yes
1.3 ~ 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 이상	2.1.1	2001-현재	PSF	yes

참고

(1) GPL-compatible doesn' t mean that we' re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-

compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don' t.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman' s lawyer has told CNRI' s lawyer that 1.6.1 is "not incompatible" with the GPL.

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

파이썬에 통합된 일부 소프트웨어에는 다른 라이센스가 적용됩니다. 라이센스는 해당 라이센스에 해당하는 코드와 함께 나열됩니다. 이러한 라이센스의 불완전한 목록은 포함된 소프트웨어에 대한 라이센스 및 승인를 참조하십시오.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

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

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

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

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

C.3 포함된 소프트웨어에 대한 라이센스 및 승인

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

C.3.1 메르센 트위스터

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

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

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

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

Redistribution and use in source and binary forms, with or without
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
modification, are permitted provided that the following conditions
are met:

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

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

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

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


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

C.3.2 소켓

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

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

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

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 비동기 소켓 서비스

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

C.3.4 쿠키 관리

`http.cookies` 모듈은 다음과 같은 주의 사항을 포함합니다:

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

               All Rights Reserved

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 실행 추적

`trace` 모듈은 다음과 같은 주의 사항을 포함합니다:

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

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

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

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

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


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

C.3.6 UUencode 및 UUdecode 함수

The `uu` codec contains the following notice:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.7 XML 원격 프로시저 호출

`xmlrpc.client` 모듈은 다음과 같은 주의 사항을 포함합니다:

```
    The XML-RPC client interface is

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

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

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

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

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

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

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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

C.3.10 SipHash24

파일 `Python/pyhash.c` 에는 Dan Bernstein의 SipHash24 알고리즘의 Marek Majkowski의 구현이 포함되어 있습니다. 여기에는 다음과 같은 내용이 포함되어 있습니다:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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

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

Original location:
   https://github.com/majek/csiphash/
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 와 dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

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

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

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

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

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

C.3.15 zlib

`zlib` 확장은 시스템에서 발견된 zlib 버전이 너무 오래되어서 빌드에 사용될 수 없으면, 포함된 zlib 소스 사본을 사용하여 빌드됩니다:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

`tracemalloc` 에 의해 사용되는 해시 테이블의 구현은 cfuhash 프로젝트를 기반으로 합니다:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

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

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

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

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

C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

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

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

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

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

C.3.18 W3C C14N 테스트 스위트

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

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

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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

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

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

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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

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

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

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD' s "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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

APPENDIX D

저작권

파이썬과 이 설명서는:

전체 라이센스 및 사용 권한 정보는 역사와 라이센스 에서 제공합니다.

색인

알파벳 이외

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z