The Python Language Reference

출시 버전 ***3.7.16***

Guido van Rossum
and the Python development team

2월 08, 2023

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 파이썬 자바구현. 이 구현은 자바 응용 프로그램을 위한 스크립트 언어로 사용되거나, 자바 클래스 라이브러리를 활용하는 응용 프로그램을 만드는데 사용될 수 있습니다. 종종 자바 라이브러리의 테스트를 만드는 데 사용되기도 합니다. 더 자세한 정보는 Jython 웹사이트 에서 찾을 수 있습니다.

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 .NET을 위한 대안 파이썬. Python.NET 과는 달리 이것은 IL을 생성하고, 파이썬 코드를 .NET 어셈블리로 직접 컴파일하는 완전한 파이썬 구현입니다. Jim Hugunin 이 만들었는데, Jython 의 원저자이기도 합니다. 자세한 정보는 IronPython 웹사이트 에서 얻을 수 있습니다.

PyPy 완전히 파이썬으로 작성된 파이썬 구현. 스택 리스(stackless) 지원이나 JIT 컴파일러와 같이 다른 구현에서는 찾을 수 없는 고급 기능을 제공합니다. 이 프로젝트의 목표 중 하나는 (파이썬으로 쓰였기 때문에) 인터프리터 수정을 쉽게 만들어서 언어 자체에 대한 실험을 북돋는 것입니다. 자세한 정보는 PyPy 프로젝트의 홈페이지 에서 찾을 수 있습니다.

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

어휘 분석과 문법의 기술은 수정된 BNF 문법 표기법을 사용합니다. 이것은 다음과 같은 정의 스타일을 사용합니다.

```
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

파이썬 프로그램은 파서*(parser)* 에 의해 읽힙니다. 파서의 입력은 어휘 분석기*(lexical analyzer)* 가 만들어내는 토큰*(token)* 들의 스트림입니다. 이 장에서는 어휘 분석기가 어떻게 파일을 토큰들로 분해하는지 설명합니다.

파이썬은 프로그램 텍스트를 유니코드 코드값으로 읽습니다; 소스 파일의 인코딩은 인코딩 선언을 통해 지정될 수 있고, 기본값은 UTF-8입니다. 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.

인코딩 선언이 발견되지 않으면 기본 인코딩은 UTF-8입니다. 여기에 더해, 파일의 처음이 UTF-8 BOM (`b'\xef\xbb\xbf'`)이면 파일 인코딩이 UTF-8으로 선언된 것으로 봅니다. (이 방식은 마이크로소프트의 **notepad** 에서 지원됩니다.)

인코딩이 선언되면, 인코딩 이름은 파이썬이 인식할 수 있어야 합니다. 인코딩은 문자열 리터럴, 주석, 식별자를 포함한 모든 어휘 분석에서 모두 사용됩니다.

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) 1~8 개의 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.

ASCII 범위 (U+0001..U+007F) 내에서, 올바른 식별자 문자는 파이썬 2.x 와 같습니다: `A` 에서 `Z` 범위의 대문자와 소문자, 밑줄 _, 첫 문자를 제외하고, 숫자 `0` 에서 `9`.

파이썬 3.0은 ASCII 범위 밖의 문자들을 도입합니다 (**PEP 3131** 참조). 이 문자들의 경우, `unicodedata` 모듈에 포함된 버전의 유니코드 문자 데이터베이스에 따라 분류됩니다.

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 un
id_continue   ::=  <all characters in id_start, plus characters in the categories Mn, M
xid_start     ::=  <all characters in id_start whose NFKC normalization is in "id_start
xid_continue  ::=  <all characters in id_continue whose NFKC normalization is in "id_co
```

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* - 하위 호환성 지원을 위해 PropList.txt 에서 명시적으로 나열된 문자들

- *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 4.1 can be found at https://www.unicode.org/Public/13.0.0/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 식별자의 예약 영역

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

_* `from module import *` 에 의해 임포트되지 않습니다. 특별한 식별자 `_` 는 대화형 인터프리터에서 마지막에 실행한 결과의 값을 저장하는 용도로 사용됩니다; `builtins` 모듈에 저장됩니다. 대화형 모드가 아닐 경우 `_` 는 특별한 의미가 없고, 정의되지도 않습니다. 임포트*(import)* 문 섹션을 보세요.

> **참고:** 이름 `_` 은 종종 국제화(internationalization)와 관련되어 사용됩니다. 이 관례에 관해서는 `gettext` 모듈의 문서를 참조하십시오.

__*__ System-defined names, informally known as “dunder” names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the 특수 메서드 이름들 section and elsewhere. More will likely be defined in future versions of Python. *Any* use of `__*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

__* 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>
```

이 생성 규칙이 보여주지 못하는 한 가지 문법적 제약은 *stringprefix* 나 *bytesprefix* 와 리터럴의 나머지 부분 사이에 공백이 허락되지 않는다는 것입니다. 소스 문자 집합은 인코딩 선언으로 정의됩니다; 소스 파일에 인코딩 선언이 없으면 UTF-8입니다. 인코딩 선언 섹션을 보세요.

쉬운 말로 하자면, 두 가지 리터럴은 한 쌍의 작은따옴표(') 나 큰따옴표(")로 둘러싸일 수 있습니다. 또한, 둘 다 한 쌍의 삼중 작은따옴표나 큰따옴표로 둘러싸일 수도 있습니다 (이것들은 보통 삼중 따옴표 된 문자열 이라고 불립니다). 역 슬래시(\) 문자는 홀로 쓰이면 특별한 의미가 있는 문자들을 이스케이핑할 때 사용되는데, 개행문자, 역 슬래시 자신, 따옴표 문자가 그것입니다.

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.

문자열과 바이트열 리터럴 모두 선택적으로 `'r'` 이나 `'R'` 문자를 앞에 붙일 수 있습니다. 이런 문자열을 날 문자열(*raw strings*) 이라고 하는데, 역 슬래시를 평범한 문자로 취급합니다. 결과적으로, 문자열 리터럴에서, 날 문자열에 있는 `'\U'` 와 `'\u'` 이스케이프는 특별하게 처리되지 않습니다. 파이썬 2.x의 날 유니코드 리터럴이 파이썬 3.x와 다르게 동작합니다는 것을 고려해서, `'ur'` 문법은 지원되지 않습니다.

버전 3.3에 추가: 날 바이트열 리터럴의 `'br'` 와 같은 의미가 있는 `'rb'` 접두어가 추가되었습니다.

버전 3.3에 추가: 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; 포맷 문자열 리터럴 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 ")

`'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	
\\	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	(1,3)
\xhh	16진수 *hh* 로 지정된 문자	(2,3)

문자열 리터럴에서만 인식되는 이스케이프 시퀀스는:

이스케이프 시퀀스	의미	유의 사항
\N{name}	유니코드 데이터베이스에서 *name* 이라고 이름 붙여진 문자	(4)
\uxxxx	16-bit 16진수 *xxxx* 로 지정된 문자	(5)
\Uxxxxxxxx	32-bit 16진수 *xxxxxxx* 로 지정된 문자	(6)

유의 사항:

(1) 표준 C와 마찬가지로, 최대 세 개의 8진수가 허용됩니다.

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

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

(4) 버전 3.3에서 변경: 별칭[1] 지원이 추가되었습니다

(5) 정확히 4개의 16진수를 필요로 합니다.

(6) 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.

> 버전 3.6에서 변경: 인식되지 않는 이스케이프 시퀀스는 DeprecationWarning 을 만듭니다. 언젠가 파이썬의 미래 버전에서는 SyntaxError 로 취급될 것입니다.

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.

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

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 역슬래시를 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 포맷 문자열 리터럴

버전 3.6에 추가.

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 | NULL | 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. 표현식 뒤로는 변환(conversion) 필드가 올 수 있는데, 느낌표 `'!'` 로 시작합니다. 포맷 지정자 (format specifier) 도 덧붙일 수 있는데, 콜론 `':'` 으로 시작합니다. 치환 필드는 닫는 중괄호 `'}'` 로 끝납니다.

포맷 문자열 리터럴의 표현식은 괄호로 둘러싸인 일반적인 파이썬 표현식으로 취급되는데, 몇 가지 예외가 있습니다. 빈 표현식은 허락되지 않고, *lambda* 표현식은 명시적인 괄호로 둘러싸야 합니다. 치환 표현식은 개행문자를 포함할 수 있으나 (예를 들어, 삼중 따옴표 된 문자열) 주석은 포함할 수 없습니다. 각 표현식은 포맷 문자열 리터럴이 등장한 지점의 문맥에서 왼쪽에서 오른쪽으로 계산됩니다.

버전 3.7에서 변경: Prior to Python 3.7, an *await* expression and comprehensions containing an *async for* clause were illegal in the expressions in formatted string literals due to a problem with the implementation.

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.

결과는 `format()` 프로토콜로 포매팅합니다. 포맷 지정자는 표현식이나 변환 결과의 *__format__()* 메서드로 전달됩니다. 포맷 지정자가 생략되면 빈 문자열이 전달됩니다. 이제 포맷된 결과가 최종 문자열에 삽입됩니다.

최상위 포맷 지정자는 중첩된 치환 필드들을 포함할 수 있습니다. 이 중첩된 필드들은 그들 자신의 변환 필드와 포맷 지정자를 포함할 수 있지만, 깊이 중첩된 치환 필드들을 포함할 수는 없습니다. 포맷 지정자 간이 언어는 문자열 .format() 메서드에서 사용되는 것과 같습니다.

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'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
```

일반적인 문자열 리터럴과 같은 문법을 공유하는 것으로 인한 결과는 치환 필드에 사용되는 문자들이 포맷 문자열 리터럴을 감싸는 따옴표와 충돌하지 않아야 한다는 것입니다:

```
f"abc {a["x"]} def"    # error: outer string literal ended prematurely
f"abc {a['x']} def"    # workaround: use different quoting
```

포맷 표현식에는 역 슬래시를 사용할 수 없고, 사용하면 에러가 발생합니다:

```
f"newline: {ord('\n')}"  # raises SyntaxError
```

역 슬래시 이스케이프가 필요한 값을 포함하려면, 임시 변수를 만들면 됩니다.

```
>>> newline = ord('\n')
>>> f"newline: {newline}"
'newline: 10'
```

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

숫자 리터럴에는 세 가지 종류가 있습니다: 정수, 실수, 허수. 복소수 리터럴 같은 것은 없습니다(복소수는 실수와 허수를 더해서 만들어집니다.)

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 실수 리터럴

실수 리터럴은 다음과 같은 어휘 정의로 표현됩니다:

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

정수부와 지수부는 항상 10진법으로 해석된다는 것에 주의해야 합니다. 예를 들어, `077e010` 는 올바른 표현이고, `77e10` 과 같은 숫자를 표현합니다. 실수 리터럴의 허락된 범위는 구현 세부 사항입니다. 정수 리터럴에서와 마찬가지로 밑줄로 숫자들의 묶음을 만드는 것도 지원됩니다.

실수 리터럴의 몇 가지 예를 듭니다:

```
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")
```

허수 리터럴은 실수부가 0.0인 복소수를 만듭니다. 복소수는 실수와 같은 범위 제약이 적용되는 한 쌍의 실수로 표현됩니다. 0이 아닌 실수부를 갖는 복소수를 만들려면, 실수를 더하면 됩니다. 예를 들어, `(3+4j)`. 허수 리터럴의 몇 가지 예를 듭니다:

```
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

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.)

모든 객체는 아이덴티티(identity), 형(type), 값(value)을 갖습니다. 객체의 아이덴티티 는 한 번 만들어진 후에는 변경되지 않습니다. 메모리상에서의 객체의 주소로 생각해도 좋습니다. '`is`' 연산자는 두 객체의 아이덴티티를 비교합니다; `id()` 함수는 아이덴티티를 정수로 표현한 값을 돌려줍니다.

CPython implementation detail: 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 값 은 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 implementation detail: 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).

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

구현이 제공하는 추적이나 디버깅 장치의 사용은 그렇지 않으면 수거될 수 있는 객체들을 살아있도록 만들 수 있음에 주의해야 합니다. 또한 '*try*…*except*' 문으로 예외를 잡는 것도 객체를 살아있게 만들 수 있습니다.

어떤 객체들은 열린 파일이나 창 같은 "외부(external)" 자원들에 대한 참조를 포함합니다. 이 자원들은 객체가 가비지 수거될 때 반납된다고 이해되지만, 가비지 수거는 보장되는 것이 아니므로, 그런 객체들은 외부자원을 반납하는 명시적인 방법 또한 제공합니다. 보통 `close()` 메서드다. 프로그램을 작성할 때는 그러한 객체들을 항상 명시적으로 닫아야(close) 합니다. '*try*…*finally*' 문과 '*with*' 문은 이렇게 하는 편리한 방법을 제공합니다.

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.

형은 거의 모든 측면에서 객체가 동작하는 방법에 영향을 줍니다. 객체의 아이덴티디가 갖는 중요성조차도 어떤 면에서는 영향을 받습니다: 불변형의 경우, 새 값을 만드는 연산은 실제로는 이미 존재하는 객체 중에서 같은 형과 값을 갖는 것을 돌려줄 수 있습니다. 반면에 가변 객체에서는 이런 것이 허용되지 않습니다. 예를 들어, `a = 1; b = 1` 후에, `a` 와 `b` 는 값 1을 갖는 같은 객체일 수도 있고, 아닐 수도 있습니다. 하지만 `c = []; d = []` 후에, `c` 와 `d` 는 두 개의 서로 다르고, 독립적이고, 새로 만들어진 빈 리스트임이 보장됩니다. (`c = d = []` 는 객은 객체를 `c` 와 `d` 에 대입합니다.)

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.

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.

NotImplemented 이 형은 하나의 값만을 갖습니다. 이 값을 갖는 하나의 객체가 존재합니다. 이 객체에는 내장된 이름 `NotImplemented` 을 통해 접근합니다. 숫자 메서드(numeric method)와 비교(rich comparison) 메서드는 제공된 피연산자에 대해 연산이 구현되지 않으면 이 값을 돌려줘야 합니다. (그러면 인터프리터는 연산자에 따라 뒤집힌 연산이나, 어떤 다른 대안을 시도합니다.) 논리값은 참입니다.

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

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

numbers.Number 이것들은 숫자 리터럴에 의해 만들어지고, 산순 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.

파이썬은 정수, 실수, 복소수를 구분합니다:

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

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.

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

numbers.Real (float) 이것들은 기계 수준의 배정도(double precision) 부동 소수점 수를 나타냅니다. 허락되는 값의 범위와 오버플로의 처리에 관해서는 하부 기계의 설계(와 C 나 자바 구현)에 따르는 수밖에 없습니다. 파이썬은 단정도(single precision) 부동 소수점 수를 지원하지 않습니다; 이것들을 사용하는 이유가 되는 프로세서와 메모리의 절감은 파이썬에서 객체를 사용하는데 들어가는 비용과 상쇄되어 미미해집니다. 그 때문에 두 가지 종류의 부동 소수점 수로 언어를 복잡하게 만들만한 가치가 없습니다.

numbers.Complex (complex) 이것들은 기계 수준 배정도 부동 소수점 수의 쌍으로 복소수를 나타냅니다. 부동 소수점 수와 한계와 문제점을 공유합니다. 복소수 `z` 의 실수부와 허수부는, 읽기 전용 어트리뷰트 `z.real` 와 `z.imag` 로 꺼낼 수 있습니다.

시퀀스들 음이 아닌 정수로 인덱싱(indexing)될 수 있는 유한한 길이의 순서 있는 집합을 나타냅니다. 내장 함수 `len()` 은 시퀀스가 가진 항목들의 개수를 돌려줍니다. 시퀀스의 길이가 n 일 때, 인덱스(index) 집합은 숫자 0, 1, …, n-1을 포함합니다. 시퀀스 a 의 항목 i 는 `a[i]` 로 선택됩니다.

시퀀스는 슬라이싱도 지원합니다: `a[i:j]` 는 $i <= k < j$ 를 만족하는 모든 항목 k 를 선택합니다. 표현식에서 사용될 때, 슬라이스는 같은 형의 시퀀스입니다. 인덱스 집합은 0에서 시작되도록 다시 번호 매겨집니다.

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) 문자열은 유니코드 코드 포인트(Unicode code point)들을 표현하는 값들의 시퀀스입니다. `U+0000 - U+10FFFF` 범위의 모든 코드 포인트들은 문자열로 표현될 수 있습니다. 파이썬에는 `char` 형이 없습니다. 대신에 문자열에 있는 각 코드 포인트는 길이 `1` 인 문자열 객체로 표현됩니다. 내장 함수 `ord()` 는 코드 포인트를 문자열 형식에서 `0 - 10FFFF` 범위의 정수로 변환합니다; `chr()` 은 범위 `0 - 10FFFF` 의 정수를 해당하는 길이 `1` 의 문자열 객체로 변환합니다. `str.encode()` 는 주어진 텍스트 인코딩을 사용해서 `str` 을 `bytes` 로 변환하고, `bytes.decode()` 는 그 반대 작업을 수행합니다.

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) 바이트열(bytes) 객체는 불변 배열입니다. 항목은 8-비트 바이트인데, 0 <= x < 256 범위의 정수로 표현됩니다. 바이트 객체를 만들 때는 바이트열 리터럴(`b'abc'` 와 같은)과 내장 `bytes()` 생성자(constructor)를 사용할 수 있습니다. 또한, 바이트열 객체는 `decode()` 메서드를 통해 문자열로 디코딩될 수 있습니다.

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

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.

확장 모듈 `array` 는 추가의 가변 시퀀스 형을 제공하는데, `collections` 모듈 역시 마찬가지입니다.

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.

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.

Dictionaries preserve insertion order, meaning that keys will be produced in the same order they were added sequentially over the dictionary. Replacing an existing key does not change the order, however removing a key and re-inserting it will add it to the end instead of keeping its old place.

딕셔너리는 가변입니다; `{...}` 표기법으로 만들 수 있습니다 (딕셔너리 디스플레이 섹션을 참고하십시오).

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

버전 3.7에서 변경: Dictionaries did not preserve insertion order in versions of Python before 3.6. In CPython 3.6, insertion order was preserved, but it was considered an implementation detail at that time rather than a language guarantee.

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.

특수 어트리뷰트들(Special attributes):

어트리뷰트	의미	
`__doc__`	함수를 설명하는 문자열 또는 없는 경우 `None`; 서브 클래스로 상속되지 않습니다.	쓰기 가능
`__name__`	함수의 이름.	쓰기 가능
`__qualname__`	함수의 정규화된 이름. 버전 3.3에 추가.	쓰기 가능
`__module__`	함수가 정의된 모듈의 이름 또는 (없는 경우) `None`	쓰기 가능
`__defaults__`	인자의 기본값 또는 (없는 경우) *None* 으로 만들어진 튜플.	쓰기 가능
`__code__`	컴파일된 함수의 바디(body) 를 나타내는 코드 객체	쓰기 가능
`__globals__`	함수의 전역 변수들을 가진 딕셔너리에 대한 참조 — 함수가 정의된 모듈의 전역 이름 공간(namespace)	읽기 전용
`__dict__`	임의의 함수 어트리뷰트를 지원하는 이름 공간.	쓰기 가능
`__closure__`	`None` 또는 함수의 자유 변수(free variable) 들에 대한 연결을 가진 셀(cell) 들의 튜플. `cell_contents` 어트리뷰트에 대한 정보는 아래를 보십시오.	읽기 전용
`__annotations__`	매개변수의 어노테이션을 가진 dict. dict의 키는 매개변수의 이름인데, 반환 값 어노테이션이 있다면 `'return'` 을 키로 사용합니다.	쓰기 가능
`__kwdefaults__`	키워드 형태로만 전달 가능한 매개변수들의 기본값을 가진 dict.	쓰기 가능

"쓰기 가능" 하다고 표시된 대부분의 어트리뷰트들은 값이 대입될 때 형을 검사합니다.

함수 객체는 임의의 어트리뷰트를 읽고 쓸 수 있도록 지원하는데, 예를 들어 함수에 메타데이터(metadata) 를 붙이는데 사용될 수 있습니다. 어트리뷰트를 읽거나 쓸 때는 일반적인 점 표현법(dot-notation) 이 사용됩니다. 현재 구현은 오직 사용자 정의 함수만 함수 어트리뷰트를 지원함에 주의해야 합니다. 내장 함수의 함수 어트리뷰트는 미래에 지원될 수 있습니다.

셀 객체는 `cell_contents` 어트리뷰트를 가지고 있습니다. 셀의 값을 읽을 뿐만 아니라 값을 설정하는 데도 사용할 수 있습니다.

함수 정의에 관한 추가적인 정보를 코드 객체로부터 얻을 수 있습니다. 아래에 나오는 내부 형의 기술을 참고하십시오.

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

특수 읽기 전용 어트리뷰트들: `__self__` 는 클래스 인스턴스 객체, `__func__` 는 함수 객체; `__doc__` 은 메서드의 설명 (`__func__.__doc__` 과 같습니다); `__name__` 은 메서드의 이름 (`__func__.__name__` 과 같습니다); `__module__` 은 메서드가 정의된 모듈의 이름이거나 없는 경우 `None`.

메서드는 기반 함수의 모든 함수 어트리뷰트들을 읽을 수 있도록 지원합니다(하지만 쓰기는 지원하지 않습니다).

어트리뷰트가 사용자 정의 함수 객체이거나 클래스 메서드 객체면, 사용자 정의 메서드 객체는 클래스의 어트리뷰트를 읽을 때 만들어질 수 있습니다 (아마도 그 클래스의 인스턴스를 통해서).

인스턴스 메서드 객체가 클래스 인스턴스를 통해 클래스의 사용자 정의 함수 객체를 읽음으로써 만들어질 때, `__self__` 어트리뷰트는 인스턴스이고, 메서드 객체는 결합(bound) 하였다고 말합니다. 새 메서드의 `__func__` 어트리뷰트는 원래의 함수 객체입니다.

클래스나 인스턴스로부터 다른 메서드 객체를 읽음으로써 사용자 정의 메서드 객체가 만들어질 때, 새 인스턴스의 `__func__` 어트리뷰트가 원래의 메서드 객체가 아니라, 그것의 `__func__` 어트리뷰트라는 점만 제외하고는 함수 객체의 경우와 같은 방식으로 동작합니다.

인스턴스 메서드 객체가 클래스나 인스턴스로부터 클래스 메서드 객체를 읽음으로써 만들어질 때, `__self__` 어트리뷰트는 클래스 자신이고, `__func__` 어트리뷰트는 클래스 메서드가 기반을 두는 함수 객체입니다.

인스턴스 메서드 객체가 호출될 때, 기반을 두는 함수 (__func__) 가 호출되는데, 인자 목록의 앞에 클래스 인스턴스 (__self__) 가 삽입됩니다. 예를 들어, C 가 함수 f() 의 정의를 포함하는 클래스이고, x 가 C 의 인스턴스일 때, x.f(1) 를 호출하는 것은 C.f(x, 1) 을 호출하는 것과 같습니다.

인스턴스 메서드 객체가 클래스 메서드 객체로부터 올 때, __self__ 에 저장된 "클래스 인스턴스" 는 실제로는 클래스 자신입니다. 그래서 x.f(1) 이나 C.f(1) 을 호출하는 것은 f(C,1) 를 호출하는 것과 같습니다 (f 는 기반 함수입니다).

함수 객체에서 인스턴스 객체로의 변환은 인스턴스로부터 어트리뷰트를 읽을 때마다 일어남에 주의해야 합니다. 어떤 경우에, 어트리뷰트를 지역 변수에 대입하고, 그 지역 변수를 호출하는 것이 효과적인 최적화가 됩니다. 또한, 이 변환이 사용자 정의 함수에 대해서만 발생함에 주의해야 합니다; 다른 콜러블 객체 (그리고 콜러블이 아닌 모든 객체) 는 변환 없이 읽힙니다. 클래스 인스턴스의 어트리뷰트인 사용자 정의 함수는 결합한 메서드로 변환되지 않는다는 것도 중요합니다; 이 변환은 함수가 클래스 어트리뷰트일 때만 일어납니다.

제너레이터 함수 (Generator functions) *yield* 문 (*yield 문* 절 참조) 을 사용하는 함수나 메서드를 제너레이터 함수 *(generator function)* 라고 부릅니다. 이런 함수를 호출하면 항상 이터레이터 (iterator) 객체를 돌려주는데, 함수의 바디 (body) 를 실행하는 데 사용됩니다: 이터레이터의 iterator.__next__() 메서드를 호출하면 yield 문이 값을 제공할 때까지 함수가 실행됩니다. 함수가 *return* 문을 실행하거나 끝에 도달하면 StopIteration 예외를 일으키고, 이터레이터는 반환하는 값들의 끝에 도달하게 됩니다.

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) *async def* 를 사용해서 정의되는 함수가 *yield* 문을 사용하면 비동기 제너레이터 함수 *(asynchronous generator function)* 라고 부릅니다. 이런 함수를 호출하면 항상 비동기 이터레이터 (asynchronous iterator) 객체를 돌려주는데, 함수의 바디 (body) 를 실행하기 위해 *async for* 문에서 사용됩니다.

비동기 이터레이터의 aiterator.__anext__() 메서드를 호출하면 어웨이터블 을 돌려주는데, await 할 때 *yield* 문이 값을 제공할 때까지 함수가 실행됩니다. 함수가 빈 *return* 문을 실행하거나 끝에 도달하면 StopAsyncIteration 예외를 일으키고, 비동기 이터레이터는 반환하는 값들의 끝에 도달하게 됩니다.

내장 함수 (Built-in functions) 내장 함수 객체는 C 함수를 둘러싸고 있습니다 (wrapper). 내장 함수의 예로는 len() 과 math.sin() (math 는 표준 내장 모듈입니다) 가 있습니다. 인자의 개수와 형은 C 함수에 의해 결정됩니다. 특수 읽기 전용 어트리뷰트들: __doc__ 은 함수의 설명 문자열 또는 없는 경우 None 입니다; __name__ 은 함수의 이름입니다; __self__ 는 None 으로 설정됩니다 (하지만 다음 항목을 보십시오); __module__ 은 함수가 정의된 모듈의 이름이거나 없는 경우 None 입니다.

내장 메서드 (Built-in methods) 이것은 사실 내장 함수의 다른 모습입니다. 이번에는 묵시적인 추가의 인자로 C 함수에 전달되는 객체를 갖고 있습니다. 내장 메서드의 예로는 alist.append() 가 있는데, *alist* 는 리스트 객체입니다. 이 경우에, 특수 읽기 전용 어트리뷰트 __self__ 는 *alist* 로 표현된 객체로 설정됩니다.

클래스 (Classes) 클래스는 콜러블입니다. 이 객체들은 보통 자신의 새로운 인스턴스를 만드는 팩토리 (factory) 로 동작하는데, *__new__()* 메서드를 재정의 (override) 하는 클래스 형에서는 달라질 수도 있습니다. 호출 인자는 *__new__()* 로 전달되고, 일반적으로, 새 인스턴스를 초기화하기 위해 *__init__()* 로도 전달됩니다.

클래스 인스턴스 (Class Instances) 클래스에서 *__call__()* 메서드를 정의함으로써, 클래스 인스턴스를 콜러블로 만들 수 있습니다.

모듈 (Modules) 모듈은 파이썬 코드의 기본적인 조직화 단위이고, *import* 문이나, importlib.import_module() 과 내장 __import__() 함수를 호출해서 구동할 수 있는 임포트 시스템 에 의해 만들어집니다. 모듈 객체는 딕셔너리 객체로 구현되는 이름 공간을 갖습니다 (이 딕셔너리 객체는 모듈에서 정의되는 함수들의 __globals__ 어트리뷰트로 참조됩니다). 어트리뷰트 참조는 이 딕셔너리에 대한 조회로 변환됩니다. 예를 들어, m.x 는 m.__dict__["x"] 와 같습니다. 모듈

객체는 모듈을 초기화하는데 사용된 코드 객체를 갖고 있지 않습니다 (일단 초기화가 끝나면 필요 없으므로).

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

미리 정의된 (쓰기 가능한) 어트리뷰트들: *__name__* 은 모듈의 이름입니다; __doc__ 은 모듈의 설명 문자열 또는 없는 경우 None 입니다; (없을 수도 있는) __annotations__ 는 모듈의 바디를 실행하면서 수집된 변수 어노테이션 들을 담은 딕셔너리입니다; *__file__* 은 모듈이 로드된 파일의 경로명입니다. 인터프리터에 정적으로 연결된 C 모듈과 같은 어떤 종류의 모듈들에서는 *__file__* 어트리뷰트가 제공되지 않습니다; 공유 라이브러리 (shared library) 로부터 동적으로 로딩되는 확장 모듈의 경우 공유 라이브러리의 경로명이 제공됩니다.

특수 읽기 전용 어트리뷰트들: __dict__ 는 딕셔너리로 표현되는 모듈의 이름 공간입니다.

CPython implementation detail: 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.

사용자 정의 클래스 (Custom classes) 사용자 정의 클래스 형들은 보통 클래스 정의 때문에 만들어집니다 (클래스 정의 섹션 참조). 클래스는 딕셔너리로 구현된 이름 공간을 갖습니다. 클래스 어트리뷰트 참조는 이 딕셔너리에 대한 조회로 변환됩니다. 예를 들어, `C.x` 는 `C.__dict__["x"]` 로 변환됩니다 (하지만 어트리뷰트에 접근하는 다른 방법들을 허락하는 여러 가지 훅 (hook) 이 있습니다.). 거기에서 어트리뷰트 이름이 발견되지 않으면, 어트리뷰트 검색은 부모 클래스들에서 계속됩니다. 이 부모 클래스 검색은 C3 메서드 결정 순서 (method resolution order) 를 사용하는데, 다중 상속이 같은 부모 클래스로 모이는 '다이아몬드 (diamond)' 계승 구조가 존재해도 올바르게 동작합니다. 파이썬이 사용하는 C3 MRO에 관한 좀 더 자세한 내용은 2.3 배포에 첨부된 문서 https://www.python.org/download/releases/2.3/mro/ 에서 찾아볼 수 있습니다.

클래스 어트리뷰트 참조가 (클래스 `C` 라고 하자) 클래스 메서드 객체로 귀결될 때는, __self__ 어트리뷰트가 `C` 인 인스턴스 메서드 객체로 변환됩니다. 스태틱 메서드로 귀결될 때는, 스태틱 메서드 객체가 감싸고 있는 객체로 변환됩니다. 클래스로부터 얻은 어트리뷰트가 __dict__ 에 저장된 값과 달라지도록 만드는 다른 방법이 디스크립터 구현하기 섹션에 나옵니다.

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).

특수 어트리뷰트들: __name__ 은 클래스의 이름입니다. __module__ 은 클래스가 정의된 모듈의 이름입니다. __dict__ 는 클래스의 이름 공간을 저장하는 딕셔너리입니다; __bases__ 는 부모 클래스들을 저장하는 튜플입니다; 부모 클래스 목록에 나타나는 순서를 유지합니다; __doc__ 은 클래스의 설명 문자열 이거나 정의되지 않으면 None 입니다; (없을 수 있는) __annotations__ 는 클래스의 바디를 실행하면서 수집된 변수 어노테이션 들을 담은 딕셔너리입니다.

클래스 인스턴스 (Class instances) 클래스 인스턴스는 클래스 객체를 호출해서 (위를 보십시오) 만들어집니다. 클래스 인스턴스는 딕셔너리로 구현되는 이름 공간을 갖는데, 어트리뷰트를 참조할 때 가장 먼저 검색되는 곳입니다. 그곳에서 어트리뷰트가 발견되지 않고, 인스턴스의 클래스가 그 이름의 어트리뷰트를 갖고 있으면, 클래스 어트리뷰트로 검색이 계속됩니다. 만약 발견된 클래스 어트리뷰트가 사용자 정의 함수면, __self__ 어트리뷰트가 인스턴스인 인스턴스 메서드로 변환됩니다. 스태틱 메서드와 클래스 메서드 객체 또한 변환됩니다. 위의 "사용자 정의 클래스 (Custom Classes)" 부분을 보십시오. 클래스로부터 얻은 어트리뷰트가 클래스의 __dict__ 에 저장된 값과 달라지도록 만드는 다른 방법이 디스크립터 구현하기 섹션에 나옵니다. 만약 클래스 어트리뷰트도 발견되지 않고, 클래스가 *__getattr__()* 메서드를 가지면, 조회를 만족시키기 위해 그 메서드를 호출합니다.

어트리뷰트 대입과 삭제는 인스턴스의 딕셔너리를 갱신할 뿐, 결코 클래스의 딕셔너리를 건드리지 않습니다. 만약 클래스가 *__setattr__()* 이나 *__delattr__()* 메서드를 가지면, 인스턴스의 딕셔너리를 갱신하는 대신에 그 메서드들을 호출합니다.

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

특수 어트리뷰트들: __dict__ 는 어트리뷰트 딕셔너리입니다; __class__ 는 인스턴스의 클래스입니다.

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.

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 코드 라고도 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: `co_name` gives the function name; `co_argcount` is the number of positional arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_cellvars` is a tuple containing the names of local variables that are referenced by nested functions; `co_freevars` is a tuple containing the names of free variables; `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals used by the bytecode; `co_names` is a tuple containing the names used by the bytecode; `co_filename` is the filename from which the code was compiled; `co_firstlineno` is the first line number of the function; `co_lnotab` is a string encoding the mapping from bytecode offsets to line numbers (for details see the source code of the interpreter); `co_stacksize` is the required stack size; `co_flags` is an integer encoding a number of flags for the interpreter.

다음과 같은 값들이 `co_flags` 를 위해 정의되어 있습니다: 함수가 가변 개수의 위치 인자를 받아들이기 위해 사용되는 `*arguments` 문법을 사용하면 비트 `0x04` 가 1이 됩니다; 임의의 키워드 인자를 받아들이기 위해 사용하는 `**keywords` 문법을 사용하면 비트 `0x08` 이 1이 됩니다; 비트 `0x20` 은 함수가 제너레이터일 때 설정됩니다.

퓨처 기능 선언 (`from __future__ import division`) 또한 코드 객체가 특정 기능이 활성화된 상태에서 컴파일되었는지를 나타내기 위해 `co_flags` 의 비트들을 사용합니다: 함수가 퓨처 division이 활성화된 상태에서 컴파일되었으면 비트 `0x2000` 이 설정됩니다; 비트 `0x10` 과 `0x1000` 는 예전 버전의 파이썬에서 사용되었습니다.

`co_flags` 의 다른 비트들은 내부 사용을 위해 예약되어 있습니다.

만약 코드 객체가 함수를 나타낸다면, `co_consts` 의 첫 번째 항목은 설명 문자열이거나 정의되지 않으면 `None` 입니다.

프레임 객체 (Frame objects) 프레임 객체는 실행 프레임(execution frame)을 나타냅니다. 트레이스백 객체에 등장할 수 있고 (아래를 보십시오), 등록된 추적 함수로도 전달됩니다.

특수 읽기 전용 어트리뷰트들: `f_back` 은 이전 스택 프레임 (호출자 방향으로)을 가리키거나, 이게 스택의 바닥이라면 `None`; `f_code` 는 이 프레임에서 실행되는 코드 객체; `f_locals` 는 지역 변수를 조회하는데 사용되는 딕셔너리; `f_globals` 는 전역 변수에 사용됩니다; `f_builtins` 는 내장된 (intrinsic) 이름들에 사용됩니다; `f_lasti` 는 정확한 바이트 코드 명령 (instruction)을 제공합니다 (코드 객체의 바이트 코드 문자열에 대한 인덱스입니다).

특수 쓰기 가능 어트리뷰트들: `f_trace` 는, `None` 이 아니면, 코드 실행 중의 여러 이벤트로 인해 호출되는 함수입니다 (디버거에서 사용됩니다). 보통 이벤트는 각 새 소스 줄에서 발생합니다 - `f_trace_lines`를 `False`로 설정하면 이것을 비활성화할 수 있습니다.

구현은 `f_trace_opcodes`를 `True`로 설정하는 것으로 요청되는 옵코드(opcode) 당 이벤트를 허용할 수 있습니다. 추적 함수에 의해 발생 된 예외가 추적되는 함수로 빠져나오면 정의되지 않은 인터프리터 동작을 유발할 수 있음에 주의해야 합니다.

`f_lineno` 는 프레임의 현재 줄 번호입니다 — 트레이스 함수(f_trace)에서 이 값을 쓰면 해당 줄로 점프합니다 (오직 가장 바닥 프레임에서만 가능합니다). 디버거는 f_lineno 를 쓰기 위한

점프 명령을 구현할 수 있습니다 (소위 Set Next Statement).

프레임 객체는 한가지 메서드를 지원합니다:

frame.**clear**()
이 메서드는 프레임이 잡은 지역 변수들에 대한 모든 참조를 제거합니다. 또한, 만약 프레임이 제너레이터에 속하면, 제너레이터가 종료됩니다(finalize). 이것은 프레임 객체가 관련된 참조 순환을 깨는 데 도움을 줍니다 (예를 들어, 예외를 잡아서 트레이스백을 추후 사용을 위해 저장할 때).

만약 프레임이 현재 실행 중이면 RuntimeError 예외가 발생합니다.

버전 3.4에 추가.

트레이스백 객체 (Traceback objects) 트레이스백 객체는 예외의 스택 트레이스를 나타냅니다. 트레이스백 객체는 예외가 발생할 때 만들어지고, types.TracebackType 를 호출해서 명시적으로 만들 수도 있습니다.

묵시적으로 만들어진 트레이스백의 경우, 예외 처리기를 찾아서 실행 스택을 되감을 때, 각각 되감기 단계마다 현재 트레이스백의 앞에 트레이스백 객체를 삽입합니다. 예외 처리기에 들어가면, 스택 트레이스를 프로그램이 사용할 수 있습니다. (*try* 문 섹션 참조.) sys.exc_info() 가 돌려주는 튜플의 세 번째 항목이나 잡힌 예외의 __traceback__ 어트리뷰트로 액세스할 수 있습니다.

프로그램이 적절한 처리기를 제공하지 않는 경우, 스택 트레이스는 표준 에러 스트림으로 (보기 좋게 포맷되어) 출력됩니다; 만약 인터프리터가 대화형이면, sys.last_traceback 으로 사용자에게 제공합니다.

명시적으로 생성된 트레이스백의 경우, tb_next 어트리뷰트를 어떻게 연결하여 전체 스택 트레이스를 형성해야 하는지를 결정하는 것은 트레이스백을 만드는 주체에게 달려 있습니다.

특수 읽기 전용 어트리뷰트들: tb_frame 은 현 단계에서의 실행 프레임입니다; tb_lineno 는 예외가 발생한 줄의 번호를 줍니다; tb_lasti 정확한 바이트 코드 명령을 가리킵니다. 만약 예외가 except 절이나 finally 절이 없는 *try* 문에서 발생하면, 줄 번호와 트레이스백의 마지막 명령 (last instruction) 은 프레임 객체의 줄 번호와 다를 수 있습니다.

특수 쓰기 가능 어트리뷰트: tb_next 는 스택 트레이스의 다음 단계 (예외가 발생한 프레임 방향으로) 이거나 다음 단계가 없으면 None 입니다.

버전 3.7에서 변경: 트레이스백 객체는 이제 파이썬 코드에서 명시적으로 인스턴스를 만들 수 있으며 기존 인스턴스의 tb_next 어트리뷰트를 변경할 수 있습니다.

슬라이스 객체 (Slice objects) 슬라이스 객체는 *__getitem__()* 메서드를 위한 슬라이스를 나타냅니다. 내장 함수 slice() 로 만들 수도 있습니다.

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) 스태틱 메서드 객체는 위에서 설명한 함수 객체를 메서드 객체로 변환하는 과정을 방지하는 방법을 제공합니다. 스태틱 메서드 객체는 다른 임의의 객체, 보통 사용자 정의 메서드를 둘러쌉니다. 스태틱 메서드가 클래스나 클래스 인스턴스로부터 읽힐 때 객체가 실제로 돌려주는 것은 둘러싸여 있던 객체인데, 다른 어떤 변환도 적용되지 않은 상태입니다. 둘러싸는 객체는 그렇더라도, 스태틱 메서드 객체 자체는 콜러블이 아닙니다. 스태틱 메서드 객체는 내장 staticmethod() 생성자로 만듭니다.

클래스 메서드 객체 (Class method objects) 스태틱 메서드 객체처럼, 클래스 메서드 객체 역시 다른 객체를 둘러싸는데, 클래스와 클래스 인스턴스로부터 그 객체를 꺼내는 방식에 변화를 줍니다.

그런 조회에서 클래스 메서드 객체가 동작하는 방식에 대해서는 위 "사용자 정의 메서드(User-defined methods)" 에서 설명했습니다. 클래스 메서드 객체는 내장 `classmethod()` 생성자로 만듭니다.

3.3 특수 메서드 이름들

클래스는 특별한 이름의 메서드들을 정의함으로써 특별한 문법 (산술 연산이나 인덱싱이나 슬라이딩 같은)에 의해 시작되는 어떤 연산들을 구현할 수 있습니다. 이것이 연산자 오버 로딩 *(operator overloading)*에 대한 파이썬의 접근법인데, 클래스가 언어의 연산자에 대해 자기 자신의 동작을 정의할 수 있도록 합니다. 예를 들어, 클래스가 *__getitem__()* 이라는 이름의 메서드를 정의하고, `x` 가 이 클래스의 인스턴스라면, `x[i]` 는 대략 `type(x).__getitem__(x, i)` 와 동등합니다. 언급된 경우를 제외하고, 적절한 메서드가 정의되지 않았을 때 연산은 예외를 일으킵니다 (보통 `AttributeError` 나 `TypeError`).

특수 메서드를 `None` 으로 설정하는 것은 해당 연산이 제공되지 않는다는 것을 가리킵니다. 예를 들어, 만약 클래스가 *__iter__()* 를 `None` 으로 설정하면, 클래스는 이터러블이 아닙니다. 따라서 이 인스턴스에 `iter()` 를 호출하면 `TypeError` 가 발생합니다. (*__getitem__()* 을 대안으로 시도하지 않습니다.)[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).

일반적인 구현은 `super().__new__(cls[, ...])` 에 적절한 인자들을 전달하는 방법으로 슈퍼 클래스의 *__new__()* 를 호출해서 새 인스턴스를 만든 후에, 돌려주기 전에 필요한 수정을 가합니다.

만약 *__new__()* 가 *cls* 의 인스턴스를 돌려준다면, 새 인스턴스의 *__init__()* 메서드가 `__init__(self[, ...])` 처럼 호출되는데, *self* 는 새 인스턴스이고, 나머지 인자들은 *__new__()* 로 전달된 것들과 같습니다.

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.

[2] *__hash__()*, *__iter__()*, *__reversed__()*, *__contains__()* 메서드들이 이런 경우에 대한 특별한 처리를 포함하고 있습니다; 다른 것들도 여전히 `TypeError` 을 일으키지만, 단지 `None` 이 콜러블이 아니므로 그런 것뿐입니다.

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.

인터프리터가 종료할 때 아직 남아있는 객체들에 대해서는 *__del__()* 메서드의 호출이 보장되지 않습니다.

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 implementation detail: 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, `<...쓸모있는 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.

이것은 디버깅에 사용되기 때문에, 표현이 풍부한 정보를 담고 모호하지 않게 하는 것이 중요합니다.

`object.`**`__str__`**`(`*`self`*`)`
: `str(object)` 와 내장 함수 `format()`, `print()` 에 의해 호출되어 객체의 "비형식적인(informal)" 또는 보기 좋게 인쇄 가능한 문자열 표현을 계산합니다. 반환 값은 반드시 문자열 객체여야 합니다.

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`*`)`
: bytes 에 의해 호출되어 객체의 바이트열 표현을 계산합니다. 반환 값은 반드시 `bytes` 객체여야 합니다.

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.

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(self), '') 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.

풍부한 비교 메서드는 주어진 한 쌍의 인자에게 해당 연산을 구현하지 않는 경우 단일자(singleton) NotImplemented 를 돌려줄 수 있습니다. 관례상, 성공적인 비교면 False 나 True 를 돌려줍니다. 하지만, 이 메서드는 어떤 형의 값이건 돌려줄 수 있습니다, 그래서 비교 연산자가 논리 문맥(Boolean context) (예를 들어 if 문의 조건)에서 사용되면, 파이썬은 결과의 참 거짓을 파악하기 위해 값에 대해 bool() 을 호출합니다.

기본적으로, *__ne__()* 는 *__eq__()* 를 호출한 후 NotImplemented 가 아니라면 그 결과를 뒤집습니다. 비교 연산자 간의 다른 암시적인 관계는 없습니다. 예를 들어, (x<y or x==y) 가 참이라고 해서 x<=y 가 참일 필요는 없습니다. 하나의 기본 연산으로부터 대소관계 연산을 자동으로 만들어 내려면 functools.total_ordering() 를 보면 됩니다.

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.

이 메서드들에 대한 (왼편의 인자는 연산을 지원하지 않지만, 오른편 인자가 지원할 때 사용되는) 뒤집힌 버전은 따로 없습니다; 대신에 *__lt__()* 와 *__gt__()* 는 서로의 뒤집힌 연산입니다; *__le__()* 와 *__ge__()* 는 서로의 뒤집힌 연산입니다; *__eq__()* 와 *__ne__()* 는 서로의 뒤집힌 연산입니다; 만약 피연산자가 서로 다른 형이고, 오른편 피연산자의 형이 왼편 피연산자의 형의 직간접적인 서브 클래스면, 오른편 피연산자의 뒤집힌 버전이 우선순위가 높습니다; 그렇지 않으면 왼편 피연산자의 메서드가 우선순위가 높습니다. 가상 서브클래싱(virtual subclassing)은 고려되지 않습니다.

object.**__hash__**(*self*)
내장 함수 hash() 와 set, frozenset, dict 와 같은 해시형 컬렉션의 멤버에 대한 연산에서 호출됩니다. *__hash__()* 는 정수를 돌려줘야 합니다. 같다고 비교되는 객체들이 같은 해시값을 가져야 한다는 성질만 요구됩니다. 객체의 비교에 사용되는 요소들로 튜플을 구성하고, 그 튜플의 해시값을 취함으로써 요소들의 해시값을 섞는 것을 권합니다. 예를 들면:

```
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.

만약 클래스가 *__eq__()* 를 정의하지 않으면 *__hash__()* 역시 정의하지 말아야 합니다. 만약 *__eq__()* 를 정의하지만 *__hash__()* 를 정의하지 않는다면, 그것의 인스턴스는 해시 가능 컬렉션에서 사용될 수 없습니다. 만약 클래스가 가변형 객체를 정의하고 있고 *__eq__()* 를 구현한다면, *__hash__()* 를 구현하지 말아야 하는데, 해시 가능 컬렉션들의 구현이 키의 해시값이 불변이도록 요구하고 있기 때문입니다(만약 객체의 해시값이 변하면, 잘못된 해시 버킷(hash bucket)에 있게 됩니다).

사용자 정의 클래스는 기본적으로 *__eq__()* 와 *__hash__()* 메서드를 갖습니다; 모든 객체는 (자기 자신을 제외하고) 같지 않다고 비교되고, `x.__hash__()` 는 적절한 값을 돌려주어, `x == y` 일 때 `x is y` 와 `hash(x) == hash(y)` 가 동시에 성립할 수 있도록 합니다.

__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, str, bytes, datetime 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.

이것은 dict 삽입의 최악의 경우 성능(worst case performance), O(n^2) 복잡도, 을 활용하기 위해 주의 깊게 선택한 입력에 의한 서비스 거부(denial-of-service) 공격에 대한 방어를 제공하기 위한 목적입니다. 자세한 내용은 http://www.ocert.org/advisories/ocert-2011-003.html 에 있습니다.

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*`)`
: 논리값 검사와 내장 연산 `bool()` 구현을 위해 호출됩니다; `False` 나 `True` 를 돌려줘야 합니다. 이 메서드가 정의되지 않는 경우, 정의되어 있다면 *__len__()* 이 호출되어, 값이 0 이 아니면 참으로 인식합니다. 만약 클래스가 *__len__()* 과 *__bool__()* 모두 정의하지 않는다면, 모든 인스턴스는 참으로 취급됩니다.

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*`)`
: 기본 어트리뷰트 액세스가 `AttributeError` 로 실패할 때 호출됩니다 (*name* 이 인스턴스 어트리뷰트 또는 `self` 의 클래스 트리에 있는 어트리뷰트가 아니라서 *__getattribute__()* 가 `AttributeError` 를 일으키거나; *name* 프로퍼티의 *__get__()* 이 `AttributeError` 를 일으킬 때). 이 메서드는 (계산된) 어트리뷰트 값을 반환하거나 `AttributeError` 예외를 일으켜야 합니다.

일반적인 메커니즘을 통해 어트리뷰트가 발견되면 *__getattr__()* 이 호출되지 않음에 주의해야 합니다 (이것은 *__getattr__()* 과 *__setattr__()* 간의 의도된 비대칭입니다). 이렇게 하는 이유는 효율 때문이기도 하고, 그렇게 하지 않으면 *__getattr__()* 가 인스턴스의 다른 어트리뷰트에 접근할 방법이 없기 때문이기도 합니다. 적어도 인스턴스 변수의 경우, 어떤 값도 인스턴스 어트리뷰트 딕셔너리에 넣지 않음으로써 (대신에 그것들을 다른 객체에 넣습니다) 완전한 제어인 것처럼 조작할 수 있습니다. 어트리뷰트 액세스를 실제로 완전히 조작하는 방법에 대해서는 아래에 나오는 *__getattribute__()* 에서 다룹니다.

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: 언어 문법이나 내장 함수에 의한 묵시적인 호출이 결과로 특수 메서드를 참조하는 경우에는 이 메서드를 거치지 않을 수 있습니다. 자세한 내용은 특수 메서드 조회 에서 다룹니다.

---

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)`

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

object.**__dir__**(*self*)
객체에 `dir()` 이 호출될 때 호출됩니다. 시퀀스를 돌려줘야 합니다. `dir()` 은 돌려준 시퀀스를 리스트로 변환한 후 정렬합니다.

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 a sequence 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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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.

버전 3.7에 추가: `__getattr__` 과 `__dir__` 모듈 어트리뷰트.

더 보기:

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

디스크립터 구현하기

다음에 오는 메서드들은 메서드를 가진 클래스(소위 디스크립터 *(descriptor)* 클래스)의 인스턴스가 소유자 *(owner)* 클래스에 등장할 때만 적용됩니다(디스크립터는 소유자 클래스의 딕셔너리나 그 부모 클래스 중 하나의 딕셔너리에 있어야 합니다). 아래의 예에서, "어트리뷰트" 는 이름이 소유자 클래스의 `__dict__` 의 키로 사용되고 있는 어트리뷰트를 가리킵니다.

`object.`**`__get__`**`(`*self*, *instance*, *owner*`)`
소유자 클래스(클래스 어트리뷰트 액세스) 나 그 클래스의 인스턴스(인스턴스 어트리뷰트 액세스)의 어트리뷰트를 취하려고 할 때 호출됩니다. *owner* 는 항상 소유자 클래스입니다. 반면에 *instance* 는 어트리뷰트 참조가 일어나고 있는 인스턴스이거나, 어트리뷰트가 *owner* 를 통해 액세스 되는 경우 *None* 입니다. 이 메서드는 (계산된) 어트리뷰트 값을 돌려주거나 `AttributeError` 예외를 일으켜야 합니다.

`object.`**`__set__`**`(`*self*, *instance*, *value*`)`
소유자 클래스의 인스턴스 *instance* 의 어트리뷰트를 새 값 *value* 로 설정할 때 호출됩니다.

`object.`**`__delete__`**`(`*self*, *instance*`)`
소유자 클래스의 인스턴스 *instance* 의 어트리뷰트를 삭제할 때 호출됩니다.

`object.`**`__set_name__`**`(`*self*, *owner*, *name*`)`
소유자 클래스 *owner* 가 만들어질 때 호출됩니다. 이 디스크립터가 *name* 에 대입되었습니다.

참고: *`__set_name__()`* is only called implicitly as part of the `type` constructor, so it will need to be called explicitly with the appropriate parameters when a descriptor is added to a class after initial creation:

```
class A:
    pass
descr = custom_descriptor()
A.attr = descr
descr.__set_name__(A, 'attr')
```

See 클래스 객체 만들기 for more details.

버전 3.6에 추가.

어트리뷰트 `__objclass__` 는 `inspect` 모듈에 의해 이 객체가 정의된 클래스를 지정하는 것으로 해석됩니다(이 값을 적절히 설정하면 동적인 클래스 어트리뷰트의 실행시간 인트로스펙션(introspection)을 지원할 수 있습니다). 콜러블의 경우, 첫 번째 위치 인자에, 주어진 형(또는 서브 클래스)의 인스턴스가 기대되거나 요구됨을 가리킬 수 있습니다(예를 들어, CPython 은 C로 구현된 연결되지 않은 메서드(unbound method)에 이 어트리뷰트를 설정합니다).

디스크립터 호출하기

일반적으로, 디스크립터는 "결합한 동작(binding behavior)"을 가진 객체 어트리뷰트입니다. 어트리뷰트 액세스가 디스크립터 프로토콜(descriptor protocol)의 메서드들에 의해 재정의됩니다: *__get__()*, *__set__()*, *__delete__()*. 이 메서드들 중 하나라도 정의되어 있으면, 디스크립터라고 부릅니다.

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 `super` 의 인스턴스에 결합하면, 결합 `super(B, obj).m()` 은 `obj.__class__.__mro__` 를 검색해서 `B` 바로 다음에 나오는 베이스 클래스 `A` 를 찾은 후에 이렇게 디스크립터를 호출합니다: `A.__dict__['m'].__get__(obj, obj.__class__)`.

인스턴스 결합의 경우, 디스크립터 호출의 우선순위는 어떤 디스크립터 메서드가 정의되어있는지에 따라 다릅니다. 디스크립터는 *__get__()*, *__set__()*, *__delete__()* 를 어떤 조합으로도 정의할 수 있습니다. 만약 *__get__()* 를 정의하지 않는다면, 어트리뷰트 액세스는, 객체의 인스턴스 딕셔너리에 값이 있지 않은 이상 디스크립터 객체 자신을 돌려줍니다. 만약 디스크립터가 *__set__()* 이나 *__delete__()* 중 어느 하나나 둘 다 정의하면, 데이터 디스크립터(data descriptor) 입니다. 둘 다 정의하지 않는다면 비데이터 디스크립터(non-data descriptor) 입니다. 보통, 데이터 디스크립터가 *__get__()* 과 *__set__()* 을 모두 정의하는 반면, 비데이터 디스크립터는 *__get__()* 메서드만 정의합니다. *__set__()* 과 *__get__()* 이 있는 데이터 디스크립터는 인스턴스 딕셔너리에 있는 값에 우선합니다. 반면에 비데이터 디스크립터는 인스턴스보다 우선순위가 낮습니다.

파이썬 메서드 (`staticmethod()` 와 `classmethod()` 를 포함해서) 는 비데이터 디스크립터로 구현됩니다. 이 때문에, 인스턴스는 메서드를 새로 정의하거나 덮어쓸 수 있습니다. 이것은 개별 인스턴스가 같은 클래스의 다른 인스턴스들과는 다른 동작을 얻을 수 있도록 만듭니다.

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

__slots__

__slots__ 은 (프로퍼티처럼) 데이터 멤버를 명시적으로 선언하고 (*__slots__* 에 명시적으로 선언하거나 부모로부터 물려받지 않는 한) *__dict__* 와 *__weakref__* 생성을 거부할 수 있도록 합니다.

__dict__ 를 사용할 때에 비교해 절약되는 공간은 상당할 수 있습니다. 어트리뷰트 조회 속도도 크게 개선할 수 있습니다.

`object.`**`__slots__`**

이 클래스 변수에는 인스턴스에 의해 사용되는 변수들의 이름을 제공하는 문자열, 이터러블(iterable), 문자열의 시퀀스가 대입될 수 있습니다. *__slots__* 은 선언된 변수들을 위한 공간을 예약하고, 간 인스턴스마다 *__dict__* 와 *__weakref__* 가 만들어지는 것을 막습니다.

__slots__ 사용에 관한 노트

- *__slots__* 가 없는 클래스를 계승할 때, 인스턴스의 *__dict__* 와 *__weakref__* 어트리뷰트는 항상 제공됩니다.
- *__dict__* 변수가 없으므로 인스턴스는 *__slots__* 정의에 나열되지 않은 새 변수를 대입할 수 없습니다. 나열되지 않은 변수명으로 대입하려고 하면 `AttributeError` 를 일으킵니다. 만약 동적으로 새 변수를 대입하는 것이 필요하다면, *__slots__* 선언의 문자열 시퀀스에 `'__dict__'` 를 추가합니다.
- 인스턴스마다 *__weakref__* 변수가 없으므로, *__slots__* 를 정의하는 클래스는 인스턴스에 대한 약한 참조(weak reference)를 지원하지 않습니다. 만약 약한 참조 지원이 필요하다면, *__slots__* 선언의 문자열 시퀀스에 `'__weakref__'` 를 추가합니다.
- *__slots__* 는 각 변수 이름마다 디스크립터를 만드는 방식으로 클래스 수준에서 구현됩니다(디스크립터 구현하기). 결과적으로, 클래스 어트리뷰트는 *__slots__* 로 정의된 인스턴스 변수들을 위한 기본값을 제공할 목적으로 사용될 수 없습니다. 클래스 어트리뷰트는 디스크립터 대입을 무효로 합니다.
- *__slots__* 선언으로 인한 효과는 그것이 정의된 클래스로 한정되지 않습니다. 부모가 선언한 *__slots__* 은 자식 클래스에 제공됩니다. 하지만, 자식 서브 클래스가 자신의 *__slots__* (새로 추가되는 변수들만 포함해야 합니다) 을 정의하지 않는다면 *__dict__* 와 *__weakref__* 를 갖게 됩니다.
- 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.
- `int`, `bytes`, `tuple`과 같은 "가변 길이(valiable-length)" 의 내장형들을 계승하는 클래스에서는 오직 빈 *__slots__* 만 지원됩니다.
- *__slots__* 에는 문자열 이외의 이터러블을 대입할 수 있습니다. 매핑도 역시 사용할 수 있습니다. 하지만, 미래에, 각 키에 대응하는 값들의 의미가 부여될 수 있습니다.
- 두 클래스가 같은 *__slots__* 을 갖는 경우만 *__class__* 대입이 동작합니다.
- 슬롯을 사용하는 여러 부모 클래스들을 다중 상속할 수 있지만, 오직 하나의 부모만 슬롯으로 만들어진 어트리뷰트를 가질 수 있습니다 (다른 베이스들은 빈 슬롯을 가져야만 합니다) - 이를 어기면 `TypeError` 를 일으킵니다.
- 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

클래스가 다른 클래스를 상속할 때, 그 클래스의 *__init_subclass__* 가 호출됩니다. 이 방법으로, 서브 클래스의 동작을 변경하는 클래스를 쓰는 것이 가능합니다. 이런 용도는 클래스 데코레이터와도 밀접히 관련되어 있습니다. 하지만 클래스 데코레이터는 그들을 사용하는 특정한 클래스에만 작용하지만, `__init_subclass__` 단독으로 그 메서드를 정의하는 클래스의 미래 서브 클래스 모두에게 작용합니다.

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.

새 클래스에 주어진 키워드 인자들은 부모 클래스의 `__init_subclass__` 로 전달됩니다. `__init_subclass__` 를 사용하는 다른 클래스들과의 호환성을 위해, 필요한 키워드 인자들을 꺼낸 후에 다른 것들을 베이스 클래스로 전달해야 합니다. 이런 식입니다:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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.

버전 3.6에 추가.

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

클래스 정의에 나타나는 베이스 클래스가 `type` 의 인스턴스가 아닌 경우, 거기에서 `__mro_entries__` 메서드를 검색합니다. 발견되면, 원래의 베이스 튜플로 호출됩니다. 이 메서드는 이 베이스 대신에 사용될 클래스의 튜플을 돌려줘야 합니다. 튜플은 비어있을 수 있습니다. 이 경우 원래 베이스는 무시됩니다.

더 보기:

PEP 560 - typing 모듈과 제네릭 형에 대한 코어 지원

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 매핑(ordered mapping) 으로 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 implementation detail: 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 3.6에서는 `DeprecationWarning`으로, 파이썬 3.8 에서는 `RuntimeError`로 이어질 것입니다.

기본 메타 클래스 type 을 사용할 때나 다른 메타 클래스가 결국 type.__new__ 를 호출할 때, 클래스 객체를 만든 후에, 다음과 같은 추가의 커스터마이제이션 단계가 실행됩니다:

- 첫째로, type.__new__ 는 *__set_name__()* 을 정의하는 클래스 이름 공간의 모든 디스크립터들을 수집합니다;
- 둘째로, 이렇게 수집된 모든 __set_name__ 을 호출하는데, 정의되고 있는 클래스와 디스크립터에 주어진 이름을 인자로 전달합니다;
- 마지막으로, 메서드 결정 순서에 따라 가장 가까운 부모에 대해 *__init_subclass__()* 훅이 호출됩니다.

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

type.__new__ 로 새 클래스가 만들어질 때, 이름 공간 매개변수로 제공되는 객체는 새로 만든 순서 있는 매핑으로 복사되고, 원래의 객체는 버립니다. 새 사본은 읽기 전용 프락시 (read-only proxy) 로 둘러싸이는데, 이것이 클래스 객체의 __dict__ 어트리뷰트가 됩니다.

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.

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

class.**__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 *__instancecheck__()* 와 *__subclasscheck__()* 를 통해 isinstance() 와 issubclass() 의 동작을 커스터마이징하는 데 필요한 규약을 포함하는데, 이 기능의 동기는 언어에 추상 베이스 클래스 (abc 모듈을 보십시오) 를 추가하고자 하는 데 있습니다.

3.3.5 제네릭 형 흉내 내기

특수 메서드를 정의함으로써 **PEP 484**에서 지정된 제네릭 클래스 문법(예를 들면 `List[int]`)을 구현할 수 있습니다:

classmethod object.**__class_getitem__**(*cls*, *key*)
key 에 있는 형 인자에 의한 제네릭 클래스의 특수화를 나타내는 객체를 돌려줍니다.

이 메서드는 클래스 개체 자체에서 조회되며, 클래스 바디에 정의된 경우, 이 메서드는 묵시적으로 클래스 메서드입니다. 이 메커니즘은 주로 정적 형 힌트와 함께 사용하기 위해 예약되어 있습니다. 다른 용도는 권장하지 않습니다.

더 보기:

PEP 560 - typing 모듈과 제네릭 형에 대한 코어 지원

3.3.6 콜러블 객체 흉내 내기

object.**__call__**(*self*[, *args...*])
인스턴스가 함수처럼 "호출될" 때 호출됩니다; 이 메서드가 정의되면, `x(arg1, arg2, ...)` 는 `x.__call__(arg1, arg2, ...)` 의 줄인 표현입니다.

3.3.7 컨테이너형 흉내 내기

The following methods can be defined to implement container objects. 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*)
내장함수 `len()` 를 구현하기 위해 호출됩니다. 객체의 길이를 돌려줘야 하는데, >= 0인 정수입니다. 또한 *__bool__()* 메서드를 정의하지 않은 객체의 *__len__()* 이 0을 돌려주면 논리 문맥에서 거짓으로 취급됩니다.

CPython implementation detail: CPython 에서, 길이는 최대 `sys.maxsize` 일 것이 요구됩니다. 만약 길이가 `sys.maxsize` 보다 크면, 어떤 기능들 (`len()` 과 같은)은 `OverflowError` 를 일으킬 수 있습니다. 참 거짓 검사에서 `OverflowError` 가 일어나는 것을 막기 위해, 객체는 *__bool__()* 를 정의해야 합니다.

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.

버전 3.4에 추가.

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*)
self[key] 의 값을 구하기 위해 호출됩니다. 시퀀스형의 경우, 정수와 슬라이스 객체만 키로 허용됩니다. 음수 인덱스(만약 클래스가 시퀀스 형을 흉내 내길 원한다면)의 특별한 해석은 *__getitem__()* 메서드에 달려있음에 주의해야 합니다. 만약 *key* 가 적절하지 않은 형인 경우, TypeError 가 발생할 수 있습니다; 만약 시퀀스의 인덱스 범위를 벗어나면(음수에 대한 특별한 해석 후에), IndexError 를 일으켜야 합니다. 매핑 형의 경우, *key* 가 (컨테이너에) 없으면, KeyError 를 일으켜야 합니다.

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

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*)
컨테이너의 이터레이터가 필요할 때 이 메서드가 호출됩니다. 이 메서드는 컨테이너에 포함된 모든 객체를 이터레이트할 수 있는 이터레이터 객체를 돌려줘야 합니다. 매핑의 경우, 컨테이너의 키를 이터레이트해야 합니다.

이터레이터 객체 역시 이 메서드를 구현할 필요가 있습니다; 자기 자신을 돌려줘야 합니다. 이터레이터 객체에 대한 추가의 정보는 typeiter 에 있습니다.

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.

The membership test operators (*in* and *not in*) are normally implemented as an iteration through a container. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be iterable.

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. *membership-test-details* 섹션을 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*)

이 메서드들은 이항 산술 연산들(+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |)을 구현하기 위해 호출됩니다. 예를 들어, *x* 가 *__add__()* 메서드를 가진 클래스의 인스턴스일 때, 표현식 `x + y` 의 값을 구하기 위해, `x.__add__(y)` 가 호출됩니다. *__divmod__()* 메서드는 *__floordiv__()* 와 *__mod__()* 를 사용하는 것과 동등해야 합니다; *__truediv__()* 와 연관되지 않아야 합니다; 내장 `pow()` 함수의 삼 항 버전이 지원되기 위해서는, *__pow__()* 메서드가 생략할 수 있는 세 번째 인자를 받도록 정의되어야 함에 주의해야 합니다.

만약 이 메서드들 중 하나가 제공된 인자에 대해 연산을 지원하지 않으면, `NotImplemented` 를 돌려줘야 합니다.

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*)

이 메서드들은 뒤집힌 피연산자들에 대해 이항 산술 연산들(+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |)을 구현하기 위해 호출됩니다. 이 함수들은 왼쪽의 피연산자가 해당 연산을 지원하지 않고[3], 피연산자들이 서로 다른 형일 때만 호출됩니다.[4] 예를 들어, 표현식 `x - y` 의 값을 구하려고 할 때, *y* 가 *__rsub__()* 를 갖는 클래스의 인스턴스이고, `x.__sub__(y)` 가 *NotImplemented* 를 돌려주면 `y.__rsub__(x)` 가 호출됩니다.

[3] 여기서 "지원하지 않는다" 는 클래스가 그런 메서드를 갖지 않거나, 메서드가 `NotImplemented` 를 돌려줌을 뜻합니다. 오른쪽 피연산자의 뒤집힌 메서드를 사용하는 대안이 시도되도록 하려면 메서드를 `None` 으로 설정하지 말아야 합니다 - 그렇게 하는 것은 그런 대안을 명시적으로 금지하는 반대 효과를 줍니다.

[4] 피연산자들이 같은 형이면, 뒤집히지 않은 메서드(*__add__()* 같은)가 실패하면 그 연산이 지원되지 않는 것으로 간주합니다. 이것이 뒤집힌 메서드가 호출되지 않는 이유입니다.

삼 항 pow() 는 *__rpow__()* 를 호출하려고 시도하지 않음에 주의해야 합니다 (그렇게 하려면 코어션 규칙이 너무 복잡해집니다).

참고: 만약 오른쪽 피연산자의 형이 왼쪽 피연산자의 형의 서브 클래스이고, 그 서브 클래스가 연산의 뒤집힌 메서드들 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*)
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*)

이 메서드들은 증분 산술 대입 (+=, -=, *=, @=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=) 을 구현하기 위해 호출됩니다. 이 메서드는 연산을 제자리에서 (*self* 를 수정해서) 하도록 시도해야 하고, 결과 (반드시 그래야 하는 것은 아니지만 *self* 일 수 있습니다) 를 돌려줘야 합니다. 만약 특정 메서드가 정의되지 않으면, 중분 대입은 일반적인 메서드들을 대신 사용합니다. 예를 들어, *x* 가 *__iadd__()* 메서드를 갖는 클래스의 인스턴스면, `x += y` 는 `x = x.__iadd__(y)` 와 동등합니다. 그렇지 않으면, `x + y` 의 값을 구할 때처럼, `x.__add__(y)` 와 `y.__radd__(x)` 가 고려됩니다. 어떤 상황에서, 증분 대입은 예상치 못한 에러로 이어질 수 있습니다. (faq-augmented-assignment-tuple-error 를 보십시오). 하지만 이 동작은 사실 데이터 모델의 일부입니다.

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.

참고: 일관성 있는 정수형 클래스를 가지려고, *__index__()* 가 정의될 때는 *__int__()* 역시 정의되어야 하고, 둘 다 같은 값을 돌려줘야 합니다.

---

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.

__int__() 가 정의되어 있지 않으면, 내장 함수 int() 는 *__trunc__()* 를 사용합니다.

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.

컨텍스트 관리자에 대한 더 자세한 정보는 typecontextmanager 에 나옵니다.

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.

  __exit__() 메서드가 전달된 예외를 다시 일으키지(reraise) 않도록 주의해야 합니다; 이것은 호출자(caller)의 책임입니다.

더 보기:

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

3.3.10 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()
```

이런 동작의 배경에 깔린 논리는, 모든 객체(형 객체를 포함해서)에 의해 구현되는 *__hash__()* 나 *__repr__()* 과 같은 많은 특수 메서드들과 관련이 있습니다. 만약 이 메서드들에 대한 묵시적인 조회가 일반적인 조회 프로세스를 거친다면, 형 객체 자체에 대해 호출되었을 때 실패하게 됩니다:

```
>>> 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

올바름을 추구하기 위해 인스턴스 어트리뷰트들을 우회하는 것에 더해, 묵시적인 특수 메서드 조회는 객체의 메타 클래스의 *__getattribute__()* 메서드 조차도 우회합니다:

```
>>> 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
```

이런 식으로 *__getattribute__()* 절차를 우회하는 것은 특수 메서드 처리의 유연함을 일부 포기하는 대신(특수 메서드가 인터프리터에 의해 일관성 있게 호출되기 위해서는 반드시 클래스 객체에 설정되어야 합니다), 인터프리터 내부에서의 속도 최적화를 위한 상당한 기회를 제공합니다.

3.4 코루틴(Coroutines)

3.4.1 어웨이터블 객체(Awaitable Objects)

어웨이터블 객체는 일반적으로 *__await__()* 메서드를 구현합니다. *async def* 함수가 돌려주는 코루틴 객체는 어웨이터블입니다.

참고: `types.coroutine()` 이나 `asyncio.coroutine()` 로 데코레이션된 제너레이터가 돌려주는 제너레이터 이터레이터 객체 또한 어웨이터블이지만 *__await__()* 를 구현하지 않습니다.

`object.`**`__await__`**`(`*self*`)`
이터레이터 를 돌려줘야 합니다. 어웨이터블 객체를 구현하기 위해 사용되어야 합니다. 예를 들어, `asyncio.Future` 는 *await* 표현식과 호환되기 위해 이 메서드를 구현합니다.

버전 3.5에 추가.

더 보기:

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

3.4.2 코루틴 객체(Coroutine Objects)

코루틴 객체는 어웨이터블 객체입니다. 코루틴의 실행은 *__await__()* 를 호출하고 그 결과를 이터레이트 하는 방법으로 제어될 수 있습니다. 코루틴이 실행을 완료하고 복귀할 때, 이터레이터는 StopIteration 을 일으키고, 예외의 value 어트리뷰트가 반환 값을 갖고 있습니다. 만약 코루틴이 예외를 일으키면, 이터레이터에 의해 퍼집니다. 코루틴이 직접 잡히지 않은 StopIteration 예외를 일으키지는 말아야 합니다.

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*)
: 코루틴의 실행을 시작하거나 재개합니다. *value* 가 None 이면, *__await__()* 가 돌려준 이터레이터를 전진시키는 것과 같습니다. *value* 가 None 이 아니면, 이 메서드는 코루틴이 일시 중지되도록 한 이터레이터의 *send()* 메서드로 위임합니다. 결과(반환 값, StopIteration 이나 다른 예외)는 위에서 설명한 *__await__()* 의 반환 값을 이터레이트할 때와 같습니다.

coroutine.**throw**(*type*[, *value*[, *traceback*]])
: 코루틴에서 지정한 예외가 발생하도록 합니다. 이 메서드는 코루틴이 일시 중지되도록 한 이터레이터의 *throw()* 메서드로 위임합니다(그런 메서드를 가지는 경우). 그렇지 않으면, 일시 중지 지점에서 예외가 발생합니다. 결과(반환 값, StopIteration 이나 다른 예외)는 위에서 설명한 *__await__()* 의 반환 값을 이터레이트할 때와 같습니다. 만약 예외가 코루틴에서 잡히지 않는다면 호출자에게 되돌아 전파됩니다.

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.

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
```

버전 3.5에 추가.

버전 3.7에서 변경: 파이썬 3.7 이전에, __aiter__ 는 비동기 이터레이터 로 결정될 어웨이터블 을 반환할 수 있었습니다.

파이썬 3.7부터, __aiter__ 는 반드시 비동기 이터레이터 객체를 돌려줘야 합니다. 다른 것을 돌려주면 TypeError 에러가 발생합니다.

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.

object.**__aenter__**(*self*)
: 이 메서드는 *__enter__()* 메서드와 의미상으로 유사한데, 유일한 차이점은 어웨이터블 을 돌려줘야 한다는 것입니다.

object.**__aexit__**(*self*, *exc_type*, *exc_value*, *traceback*)
: 이 메서드는 *__exit__()* 메서드와 의미상으로 유사한데, 유일한 차이점은 어웨이터블 을 돌려줘야 한다는 것입니다.

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')
```

버전 3.5에 추가.

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. 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.

다음과 같은 것들이 이름을 연결합니다: 함수로 전달되는 형식 매개변수, *import* 문, 클래스와 함수 정의(이것들은 클래스나 함수 이름을 정의하고 있는 블록에 연결합니다), 그리고 다음과 같은 것들에 등장하는 식별자 대상들: 대입, *for* 루프 헤더, *with* 문이나 *except* 절의 `as` 뒤. `from ... import *` 형태의 `import` 문은 임포트되는 모듈에 정의된 모든 이름을 연결합니다, 밑줄로 시작하는 이름들은 예외입니다. 이 형태는 모듈 수준에서만 사용될 수 있습니다.

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.

만약 이름이 블록 내에서 연결되면, *nonlocal* 이나 *global* 로 선언되지 않는 이상, 그 블록의 지역 변수입니다. 만약 이름이 모듈 수준에서 연결되면, 전역 변수입니다. (모듈 코드 블록의 변수들 지역이면서 전역입니다.) 만약 변수가 코드 블록에서 사용되지만, 거기에서 정의되지 않았으면 자유 변수 *(free variable)* 입니다.

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.

만약 이름 연결 연산이 코드 블록 내의 어디에서 건 일어난다면, 그 블록 내에서 그 이름의 모든 사용은 현재 블록을 가리키는 것으로 취급됩니다. 이것은 연결되기 전에 블록에서 사용될 때 에러로 이어질 수 있습니다. 이 규칙은 미묘합니다. 파이썬에는 선언(declaration)이 없고, 이름 연결 연산이 코드 블록 내의 어디에서나 일어날 수 있도록 허락합니다. 코드 블록의 지역 변수는 블록의 텍스트 전체에서 이름 연결 연산을 찾아야 결정될 수 있습니다.

만약 *global* 문이 블록 내에서 나오면, 문장에서 지정한 이름의 모든 사용은 최상위 이름 공간(top-level namespace)에 연결된 것을 가리키게 됩니다. 최상위 이름 공간에서 이름을 검색한다는 것은, 전역 이름 공간, 즉 코드 블록을 포함하는 모듈의 이름 공간, 과 내장 이름 공간, 모듈 `builtins` 의 이름 공간, 을 검색한다는 뜻입니다. 전역 이름 공간이 먼저 검색됩니다. 거기에서 이름이 발견되지 않으면, 내장 이름 공간을 검색합니다. `global` 문은 그 이름을 사용하기 전에 나와야 합니다.

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.

nonlocal 문은 대응하는 이름이 가장 가까이서 둘러싸는 함수 스코프에서 이미 연결된 이름을 가리키도록 만듭니다. 만약 주어진 이름이 둘러싸는 함수 스코프 어디에도 없다면 컴파일 시점에 `SyntaxError` 를 일으킵니다.

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

클래스 정의 블록과 `exec()` 와 `eval()` 로 전달되는 인자는 특별한 이름 검색 문맥을 갖습니다. 클래스 정의는 이름을 사용하고 정의할 수 있는 실행 가능한 문장입니다. 이 참조들은 연결되지 않은 지역 변수를 전역 이름 공간에서 찾는다는 점을 제외하고는 이름 검색의 일반적인 규칙을 따릅니다. 클래스 정의의 이름 공간은 클래스의 어트리뷰트 딕셔너리가 됩니다. 클래스 블록에서 정의된 이름들의 스코프는 클래스 블록으로 제한됩니다; 메서드들의 코드 블록으로 확대되지 않습니다 – 이것은 컴프리헨션과 제너레이터 표현을 포함하는데 이것들이 함수 스코프를 사용해서 구현되기 때문입니다. 이것은 다음과 같은 것이 실패한다는 뜻입니다:

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

4.2.3 builtins 와 제한된 실행

CPython implementation detail: 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.

4.2.4 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.

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.

예외는 클래스 인스턴스로 구분됩니다. *except* 절은 인스턴스의 클래스에 따라 선택됩니다: 인스턴스의 클래스나 그것의 베이스 클래스를 가리켜야 합니다. 인스턴스는 핸들러가 수신할 수 있고 예외적인 조건에 대한 추가적인 정보를 포함할 수 있습니다.

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.

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

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).

[1] `types.ModuleType` 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.

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.

모든 모듈은 이름이 있습니다. 서브 패키지 이름은 파이썬의 표준 어트리뷰트 액세스 문법을 따라, 부모 패키지 이름과 점(dot)으로 구분됩니다. 그래서 sys 라고 불리는 모듈과 email 이라고 불리는 패키지가 있을 수 있습니다. email 은 다시 서브 패키지 email.mime 을 갖고, 이 서브 패키지 내에 모듈 email.mime.text 가 있을 수 있습니다.

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

5.3.4 메타 경로(meta path)

주어진 이름의 모듈을 sys.modules 에서 찾을 수 없을 때, 파이썬은 sys.meta_path 를 검색하는데, 메타 경로 파인더 객체들의 목록을 포함하고 있습니다. 이 파인더들이 주어진 이름의 모듈을 처리하는 방법을 알고 있는지 확인하도록 요청합니다. 메타 경로 파인더들은 find_spec() 라는 이름의 메서드를 구현해야만 하는데, 세 개의 인자를 받아들입니다: 이름, 임포트 경로, (생략 가능한) 타깃(target) 모듈. 메타 경로 파인더는 주어진 이름의 모듈을 처리할 수 있는지를 결정하기 위해 어떤 전략이건 사용할 수 있습니다.

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.

메타 경로 파인더의 find_spec() 메서드는 두 개나 세 개의 인자로 호출됩니다. 첫 번째 인자는 모듈의 완전히 정규화된 이름(fully qualified name)입니다, 예를 들어 foo.bar.baz. 두 번째 인자는 모듈 검색에 사용할 경로 엔트리입니다. 최상위 모듈이 경우 두 번째 인자는 None 이지만, 서브 모듈이나 서브 패키지의 경우 두 번째 인자는 부모 패키지의 __path__ 어트리뷰트 값입니다. 만약 적절한 __path__ 어트리뷰트를 참조할 수 없으면 ModuleNotFoundError 를 일으킵니다. 세 번째 인자는 이미 존재하는 모듈 객체인데, 뒤에서 로딩할 대상이 됩니다. 임포트 시스템은 다시 로드(reload) 할 때만 타깃을 전달합니다.

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) 들

에 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: 메타 경로 파인더의 `find_spec()` 메서드가 이제 폐지된 `find_module()` 을 대체합니다. 변경 없이도 동작하기는 하지만, 임포트 절차는 파인더가 `find_spec()` 을 구현하지 않았을 때만 `find_module()` 을 사용합니다.

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:
    if spec.submodule_search_locations is not None:
        # namespace package
        sys.modules[spec.name] = module
    else:
        # unsupported
        raise ImportError
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
    # Set __loader__ and __package__ if missing.
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.

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.

버전 3.4에 추가: 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.

[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.

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.

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
    bar.py
```

그리고 spam/__init__.py 가 다음과 같은 줄들을 포함한다고 합시다:

```
from .foo import Foo
from .bar import Bar
```

그러면 다음과 같이 실행하면 spam 모듈에 foo 와 bar 에 대한 이름 연결이 일어납니다.

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

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 모듈 스펙

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.

모듈의 스펙은 모듈 객체의 __spec__ 어트리뷰트로 노출됩니다. 모듈 스펙의 내용에 대한 세부 사항은 ModuleSpec 을 보세요.

버전 3.4에 추가.

5.4.4 임포트 관련 모듈 어트리뷰트

임포트 절차는 로딩하는 동안 로더가 모듈을 실행하기 전에 모듈의 스팩에 기초해서 각 모듈 객체에 이 어트리뷰트들을 채워 넣습니다.

__name__
: __name__ 어트리뷰트는 모듈의 완전히 정규화된(fully-qualified) 이름으로 설정되어야 합니다. 이 이름은 임포트 시스템이 모듈을 유일하게(uniquely) 식별하는 데 사용됩니다.

__loader__
: __loader__ 어트리뷰트는 모듈을 로드할 때 임포트 절차가 사용한 로더 객체로 설정되어야 합니다. 이것은 주로 인트로스펙션(introspection)을 위한 것이지만, 추가적인 로더에 국한된 기능들을 위한 것이기도 합니다, 예를 들어 로더와 결합한 데이터를 얻는 것이 있습니다.

__package__
: 모듈의 __package__ 어트리뷰트는 반드시 설정되어야 합니다. 값은 문자열이어야 하는데, __name__ 과 같은 값일 수 있습니다. 모듈이 패키지일 때, __package__ 값은 __name__ 으로 설정되어야 합니다. 모듈이 패키지가 아닐 때, 최상위 모듈이면 빈 문자열로 설정되고, 서브 모듈이면 부모 패키지의 이름으로 설정되어야 합니다. 더 상세한 내용은 **PEP 366** 을 참조하세요.

  PEP 366 에 정의되어 있듯이, 메인 모듈에서 명시적인 상대 임포트를 계산할 때, __name__ 대신, 이 어트리뷰트가 사용됩니다. __spec__.parent 과 같은 값일 것으로 기대됩니다.

  버전 3.6에서 변경: __package__ 의 값이 __spec__.parent 과 같을 것으로 기대됩니다.

__spec__
: __spec__ 어트리뷰트는 모듈을 임포트 할 때 사용한 모듈 스펙으로 설정되어야 합니다. __spec__ 을 적절히 설정하는 것은 인터프리터가 구동되는 동안 초기화되는 모듈들 에도 마찬가지로 적용됩니다. 한가지 예외는 __main__ 인데, 어떤 경우에 __spec__ 이 어떤 경우에 *None* 으로 설정됩니다.

  __package__ 가 정의되지 않으면, 대체물로 __spec__.parent 가 사용됩니다.

  버전 3.4에 추가.

  버전 3.6에서 변경: __package__ 가 정의되지 않으면, 대체물로 __spec__.parent 가 사용됩니다.

__path__
: 모듈이 패키지면 (정규 또는 이름 공간), 모듈 객체의 __path__ 어트리뷰트가 반드시 설정되어야 합니다. 값은 이터러블이어야 하는데, __path__ 가 더는 의미가 없으면 빈 이터러블일 수 있습니다. 만약 __path__ 가 비어있지 않다면, 탐색할 때 문자열을 제공해야 합니다. __path__ 의 의미에 관한 자세한 내용은 아래에 나옵니다.

  패키지가 아닌 모듈은 __path__ 어트리뷰트가 없어야 합니다.

__file__

__cached__
: __file__ 은 생략될 수 있다. 만약 설정되면, 이 어트리뷰트의 값은 문자열이어야 합니다. 임포트 시스템은 의미가 없을 때(예를 들어 데이터베이스에서 로드된 모듈) __file__ 을 설정하지 않을 수 있습니다.

  만약 __file__ 이 설정되면, __cached__ 역시 설정하는 것이 적절할 수 있는데, 코드의 컴파일된 버전(예를 들어, 바이트 컴파일된 파일)을 가리키는 경로입니다. 이 어트리뷰트를 설정하기 위해 파일이 꼭 존재해야 할 필요는 없습니다; 경로는 단순히 컴파일된 파일이 있어야 할 곳을 가리킬 수 있습니다(**PEP 3147** 을 보세요).

  __file__ 이 설정되지 않을 때도, __cached__ 를 설정하는 것이 적절할 수 있습니다. 하지만, 그런 시나리오는 아주 예외적입니다. 궁극적으로, 로더가 __file__ 이나 __cached__ 혹은 둘 모두를 사용합니다. 그래서 로더가 캐싱된 모듈을 로드할 수는 있지만, 파일로부터 직접 로드할 수 없다면, 예외적인 시나리오가 적절할 수 있습니다.

5.4.5 module.__path__

정의에 따르면, 모듈에 __path__ 어트리뷰트가 있으면, 이 모듈은 패키지입니다.

패키지의 __path__ 어트리뷰트는 서브 패키지를 로딩할 때 사용합니다. 임포트 절차 내에서, 임포트하는 동안 모듈을 검색할 위치들의 목록을 제공한다는 점에서 sys.path 와 같은 기능을 갖습니다. 하지만 __path__ 는 보통 sys.path 보다 제약 조건이 많습니다.

__path__ 는 문자열의 이터러블이지만, 비어있을 수 있습니다. sys.path 과 같은 규칙이 패키지의 __path__ 에도 적용되고, 패키지의 __path__ 를 탐색하는 동안 sys.path_hooks (아래에서 설명한다)에게 의견을 묻습니다.

패키지의 __init__.py 파일은 패키지의 __path__ 어트리뷰트를 설정하거나 변경할 수 있고, 이것이 **PEP 420** 이전에 이름 공간 패키지를 구현하는 방법으로 사용됐습니다. **PEP 420** 의 도입으로 인해, 이름 공간 패키지가 __path__ 조작 코드만을 포함하는 __init__.py 파일을 제공할 필요가 없어졌습니다; 임포트 절차가 자동으로 이름 공간 패키지를 위한 __path__ 를 설정합니다.

5.4.6 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 3.4에서 변경: loader.module_repr() 의 사용이 폐지되었고 이제 모듈 repr 를 만드는데 임포트 절차에 의해 모듈 스펙이 사용됩니다.

파이썬 3.3과의 과거 호환성을 위해, 위에서 설명한 방법들을 시도하기 전에, 만약 정의되어 있으면, 로더의 module_repr() 메서드를 호출해서 모듈 repr 을 만듭니다. 하지만, 그 메서드는 폐지되었습니다.

5.4.7 캐시된 바이트 코드 무효화

Before Python loads cached bytecode from .pyc file, it checks whether the cache is up-to-date with the source .py file. By default, Python does this by storing the source’s last-modified timestamp and size in the cache file when writing it. At runtime, the import system then validates the cache file by checking the stored metadata in the cache file against the source’s metadata.

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 는 모듈과 패키지의 검색 위치를 제공하는 문자열의 목록을 포함합니다. PYTHONPATH 환경 변수와 여러 가지 설치와 구현 특정 기본값들로부터 초기화됩니다. sys.path 에 있는 엔트리들은 파일 시스템의 디렉터리와 zip 파일을 가리키고, 그밖에 잠재적으로 모듈 검색에 사용될 수 있는 "장소들"(site 모듈을 보라)을 가리킬 수 있는데, URL이나 데이터베이스 조회 같은 것들입니다. sys.path 에는 문자열과 바이트열만 있어야 합니다; 다른 모든 형은 무시됩니다. 바이트열의 인코딩은 개별 경로 엔트리 파인더들에 의해 결정됩니다.

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.

경로 기반 파인더는 검색 경로의 모든 엔트리를 탐색하고, 개별 엔트리마다 적절한 경로 엔트리 파인더 (PathEntryFinder)를 찾습니다. 이것은 비용이 많이 드는 연산일 수 있으므로 (예를 들어, 이 검색을 위해 *stat()* 호출로 인한 부하가 있을 수 있습니다), 경로 기반 파인더는 경로 엔트리를 경로 엔트리 파인더로 매핑하는 캐시를 관리합니다. 이 캐시는 sys.path_importer_cache 에 유지됩니다 (이름에도 불구하고, 이 캐시는 임포터 객체로 제한되지 않고 실제로는 파인더 객체를 저장합니다). 이런 방법으로, 특정 경로 엔트리 위치의 경로 엔트리 파인더 의 비싼 검색은 오직 한 번만 수행됩니다. 사용자 코드가

sys.path_importer_cache 의 캐시 엔트리를 삭제해서 경로 기반 파인더가 그 경로 엔트리를 다시 검색하도록 하는 것이 허락됩니다[3].

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.

현재 작업 디렉터리(current working directory) – 빈 문자열로 표현된다 – 는 sys.path 에 있는 다른 엔트리들과 약간 다르게 취급됩니다. 첫째로, 현재 작업 디렉터리가 존재하지 않음이 발견되면 sys.path_importer_cache 에는 아무런 값도 저장되지 않습니다. 둘째로, 현재 작업 디렉터리는 각 모듈 조회 때마다 다시 확인됩니다. 셋째로, sys.path_importer_cache 에 사용되는 경로와 importlib.machinery.PathFinder.find_spec() 가 돌려주는 경로는 빈 문자열이 아니라 실제 현재 작업 디렉터리가 됩니다.

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() takes two arguments: the fully qualified name of the module being imported, and the (optional) target module. find_spec() returns a fully populated spec for the module. This spec will always have "loader" set (with one exception).

To indicate to the import machinery that the spec represents a namespace *portion*, the path entry finder sets "loader" on the spec to None and "submodule_search_locations" to a list containing the portion.

버전 3.4에서 변경: find_spec() 이 find_loader() 와 find_module() 를 대체하는데, 둘 다 이제 폐지되었습니다, find_spec() 이 정의되지 않으면 이것들을 사용합니다.

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() 는 하나의 인자를 받아들입니다, 임포트되는 모듈의 완전히 정규화된 이름. find_loader() 는 2-튜플을 돌려주는데, 첫 번째 항목은 로더이고 두 번째 항목은 이름 공간 포션 입니다. 첫 번째 항목(즉 로더)이 None 이면, 경로 엔트리 파인더가 주어진 이름의 모듈에 대한 로더를 제공하지는 못하지만, 경로 엔트리가 주어진 이름의 모듈에 대한 이름 공간 포션에 이바지함을 안다는 뜻입니다. 이것은 거의 항상, 파이썬이 파일 시스템에 물리적으로 존재하지 않는 이름 공간 패키지를 임포트하도록 요구되는 경우입니다. 경로 엔트리 파인더가 로더로 None 을 돌려줄 때, 2-튜플의 두 번째 항목은 시퀀스여야 하는데 비어있을 수도 있습니다.

find_loader() 가 None 이 아닌 로더 값을 돌려주면, 그 포션은 무시되고 경로 기반 파인더가 로더를 돌려주며 경로 엔트리 검색을 종료합니다.

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

[3] 예전 코드에서, sys.path_importer_cache 에서 imp.NullImporter 의 인스턴스를 찾는 것이 가능합니다. 코드가 대신 None 을 사용하도록 변경할 것을 권고합니다. 더 자세한 내용은 portingpythoncode 를 참조하세요.

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.

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.

To selectively prevent the import of some modules from a hook early on the meta path (rather than disabling the standard import system entirely), it is sufficient to raise ModuleNotFoundError directly from find_spec() instead of returning None. The latter indicates that the meta path search should continue, while raising an exception terminates it immediately.

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.py나 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.ZZZ를 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 열린 이슈들

XXX 도표가 있으면 정말 좋겠다.

XXX * (import_machinery.rst) 모듈과 패키지의 어트리뷰트들에만 할당된 섹션은 어떨까? 아마도 데이터 모델 레퍼런스 페이지에 있는 관련 항목들을 확장하거나 대체해야 할 것이다.

XXX 라이브러리 설명서의 runpy, pkgutil 등등은 새 임포트 시스템 섹션으로 가는 “See Also” 링크를 처음에 붙여야만 한다.

XXX __main__ 이 초기화되는 다른 방법들에 대한 설명을 더 붙여야 하나?

XXX __main__ 의 까다로움/어려움에 대한 정보를 추가하자 (즉 **PEP 395** 의 사본)

5.10 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 은 파이썬 3.3 에 이름 공간 패키지 를 도입했습니다. PEP 420은 `find_module()` 의 대안으로 `find_loader()` 프로토콜 역시 도입했습니다.

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 type", this means that the operator implementation for built-in types works as follows:

- 어느 한 인자가 복소수면 다른 하나는 복소수로 변환됩니다;
- 그렇지 않고, 어느 한 인자가 실수면, 다른 하나는 실수로 변환됩니다;
- 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.

비공개 이름 뒤섞기(private name mangling): 클래스 정의에 등장하는 식별자가 두 개나 그 이상의 밑줄로 시작하고, 두 개나 그 이상의 밑줄로 끝나지 않으면, 그 클래스의 비공개 이름*(private name)* 으로 간주합니다. 비공개 이름은 그 들을 위한 코드가 만들어지기 전에 더 긴 형태로 변환됩니다. 이 변환은 그 이름의 앞에 클래스 이름을 삽입하는데, 클래스 이름의 처음에 오는 모든 밑줄을 제거한 후, 하나의 밑줄을 추가합니다. 예를 들어, `Ham` 이라는 이름의 클래스에 식별자 `__spam` 이 등장하면, `_Ham__spam` 으로 변환됩니다. 이 변환은 식별자가 사용되는 문법적인 문맥에 무관합니다. 변환된 이름이 극단적으로 길면(255자보다 길면), 구현이 정의한 잘라내기가 발생할 수 있습니다. 클래스 이름이 밑줄로만 구성되어 있으면, 변환은 일어나지 않습니다.

6.2.2 리터럴 (Literals)

파이썬은 문자열과 바이트열 리터럴과 여러 가지 숫자 리터럴들을 지원합니다:

```
literal   ::=   stringliteral | bytesliteral
                | integer | floatnumber | imagnumber
```

리터럴의 값을 구하면 주어진 형(문자열, 바이트열, 정수, 실수, 복소수)과 주어진 값을 갖는 객체가 나옵니다. 실수와 복소수의 경우는 근삿값일 수 있습니다. 자세한 내용은 리터럴 섹션을 보세요.

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).

튜플이 괄호에 의해 만들어지는 것이 아니라, 쉼표 연산자의 사용 때문이라는 것에 주의해야 합니다. 예외는 빈 튜플인데, 괄호가 필요합니다 — 표현식에서 괄호 없는 "없음(nothing)"을 허락하는 것은 모호함을 유발하고 자주 발생하는 오타들이 잡히지 않은 채로 남게 할 것입니다.

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  ::=  expression comp_for
comp_for       ::=  ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter      ::=  comp_for | comp_if
comp_if        ::=  "if" expression_nocond [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 (Python 3.7에서, 이러한 표현식은 컴파일될 때 `DeprecationWarning` 을 발생시킵니다. 파이썬 3.8+ 에서는 `SyntaxError` 를 일으킬 것입니다).

파이썬 3.6부터, *`async def`* 함수에서는, 비동기 이터레이터 를 탐색하기 위해 `async for` 를 사용할 수 있습니다. `async def` 함수에 있는 컴프리헨션은 처음에 나오는 표현식 뒤에 `for` 나 `async for` 절이 올 수 있고, 추가의 `for` 나 `async for` 절이 올 수 있고, *`await`* 표현식 또한 사용할 수 있습니다. 컴프리헨션이 `async for` 절이나 `await` 표현식을 포함하면 비동기 컴프리헨션 *(asynchronous comprehension)* 이라고 불립니다. 비동기 컴프리헨션은 그것이 등장한 코루틴 함수의 실행을 일시 중지시킬 수 있습니다. **PEP 530** 를 참조하세요.

버전 3.6에 추가: 비동기 컴프리헨션이 도입되었습니다.

버전 3.7부터 폐지: `yield` 와 `yield from` 은 묵시적으로 중첩된 스코프에서 폐지되었습니다.

6.2.5 리스트 디스플레이

리스트 디스플레이는 대괄호(square brackets)로 둘러싸인 표현식의 나열인데 비어있을 수 있습니다:

```
list_display  ::=  "[" [starred_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  ::=  "{" (starred_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

딕셔너리 디스플레이는 중괄호(curly braces)로 둘러싸인 키/데이터 쌍의 나열인데 비어있을 수 있습니다:

```
dict_display        ::=  "{" [key_datum_list | dict_comprehension] "}"
key_datum_list      ::=  key_datum ("," key_datum)* [","]
key_datum           ::=  expression ":" expression | "**" or_expr
dict_comprehension  ::=  expression ":" expression comp_for
```

딕셔너리 디스플레이는 새 딕셔너리 객체를 만듭니다.

쉼표로 분리된 키/데이터 쌍의 시퀀스가 주어질 때, 그것들은 왼쪽에서 오른쪽으로 값이 구해지고 딕셔너리의 엔트리들을 정의합니다: 각 키 객체는 딕셔너리에 대응하는 데이터를 저장하는 데 키로 사용됩니다. 이것은 키/값 목록에서 같은 키를 여러 번 지정할 수 있다는 뜻인데, 그 키의 최종 딕셔너리 값은 마지막에 주어진 것이 됩니다.

두 개의 애스터리스크(asterisk) `**` 는 딕셔너리 언 패킹 *(dictionary unpacking)* 를 나타냅니다. 피연산자는 매핑 이어야만 합니다. 각 매핑 항목은 새 딕셔너리에 추가됩니다. 뒤에 오는 값들이 앞의 키/데이터 쌍이나 앞의 딕셔너리 언 패킹 때문에 설정된 값들을 교체합니다.

버전 3.5에 추가: **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.

킷값의 형에 대한 제약은 앞의 섹션 표준형 계층 에서 나열되었습니다. (요약하자면, 키 형은 해시 가능해야 하는데, 모든 가변 객체들이 제외됩니다.) 중복된 키 간의 충돌은 감지되지 않습니다; 주어진 키에 대해 저장된 마지막 (구문상으로 디스플레이의 가장 오른쪽에 있는) 데이터가 우선합니다.

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 (Python 3.7 에서, 이러한 표현식은 컴파일될 때 `DeprecationWarning` 을 발생시킵니다. 파이썬 3.8+ 에서는 `SyntaxError` 를 일으킬 것입니다).

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).

버전 3.6에 추가: 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 3.7부터 폐지: `yield` 와 `yield from` 은 묵시적으로 중첩된 스코프에서 폐지되었습니다.

6.2.9 일드 표현식 (Yield expressions)

```
yield_atom       ::=  "(" yield_expression ")"
yield_expression ::=  "yield" [expression_list | "from" expression]
```

일드 표현식은 제너레이터 함수나 비동기 제너레이터 함수를 정의할 때 사용되고, 그래서 함수 정의의 바디에서만 사용될 수 있습니다. 함수의 바디에서 일드 표현식을 사용하는 것은 함수를 제너레이터로 만들고, *`async def`* 함수의 바디에서 사용하는 것은 그 코루틴 함수를 비동기 제너레이터로 만듭니다. 예를 들어:

```
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 (Python 3.7 에서, 이러한 표현식은 컴파일될 때 `DeprecationWarning` 을 발생시킵니다. 파이썬 3.8+ 에서는 `SyntaxError` 를 일으킬 것입니다).

버전 3.7부터 폐지: 일드 표현식은 컴프리헨션과 제너레이터 표현식을 구현하는 데 사용되는 묵시적으로 정의된 스코프에서 폐지되었습니다.

제너레이터 함수는 다음에서 설명합니다. 반면에 비동기 제너레이터 함수는 비동기 제너레이터 함수 섹션에서 별도로 설명합니다.

제너레이터 함수가 호출될 때, 제너레이터로 알려진 이터레이터를 돌려줍니다. 그러면 그 제너레이터가 제너레이터 함수의 실행을 제어합니다. 제너레이터의 메서드들 중 하나가 호출될 때 실행이 시작됩니다. 그 시점에, 실행은 첫 번째 일드 표현식까지 진행한 후, 거기에서 다시 일시 중지 (suspend) 하고 제너레이터의 호출자에게 *`expression_list`* 의 값을 돌려줍니다. 일시 중지된다는 것은, 모든 지역 상태가 보존된다는 뜻인데, 지역 변수들의 현재 연결들, 명령 포인터 (instruction pointer), 내부 연산 스택 (internal evaluation stack), 모든 예외 처리 상태가 포함됩니다. 제너레이터의 메서드들 중 하나를 호출해서 실행이 재개될 때, 함수는 마치 일드 표현식이 단지 또 하나의 외부 호출인 것처럼 진행할 수 있습니다. 재개된 후에 일드 표현식의 값은 실행을 재개하도록 만든 메서드에 달려있습니다. (보통 *`for`* 나 `next()` 내장을 통해) *`__next__()`* 가 사용되었다면 결과는 `None` 입니다. 그렇지 않고, *`send()`* 가 사용되었다면, 결과는 그 메서드로 전달된 값입니다.

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 표현식을 서브 이터레이터(subiterator)로 취급합니다. 서브 이터레이터가 만드는 모든 값은 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 `yield_from` 문법을 도입해서, 서브 제너레이터로의 위임을 쉽게 만드는 제안.

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__`**`()`
: 제너레이터 함수의 실행을 시작하거나 마지막으로 실행된 일드 표현식에서 재개합니다. 제너레이터 함수가 *__next__()* 메서드로 재개될 때, 현재의 일드 표현식은 항상 `None` 값을 갖는다. 실행은 다음 일드 표현식까지 이어지는데, 그곳에서 제너레이터는 다시 일시 중지되고, *expression_list* 의 값을 *__next__()* 의 호출자에게 돌려줍니다. 제너레이터가 다른 값을 yield 하지 않고 종료되면 `StopIteration` 예외가 발생합니다.

  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`**`(`*type*`[`, *value*`[`, *traceback*`]]`)
: 제너레이터가 일시 정지한 지점에서 `type` 형의 예외를 일으키고, 제너레이터 함수가 yield 하는 다음 값을 돌려줍니다. 제너레이터가 다른 값을 yield 하지 않고 종료하면 `StopIteration` 을 일으킵니다. 제너레이터가 전달된 예외를 잡지 않거나, 다른 예외를 일으키면, 그 예외는 호출자로 퍼집니다.

`generator.`**`close`**`()`
: 제너레이터가 일시 정지한 지점에서 `GeneratorExit` 를 일으킵니다. 그런 다음 제너레이터 함수가 우아하게(gracefully) 종료하거나, 이미 닫혔거나, (그 예외를 잡지 않음으로써) `GeneratorExit` 를 일으키면 close는 호출자로 돌아갑니다. 제너레이터가 값을 yield 하면 `RuntimeError` 가 발생합니다. 제너레이터가 다른 예외를 일으키면, 호출자로 퍼집니다. 제너레이터가 예외나 정상 종료로 인해 이미 종료되었다면, *close()* 는 아무런 일도 하지 않습니다.

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.

비동기 제너레이터의 메서드들 중 하나를 호출하면 어웨이터블 객체를 돌려주고, 이 객체를 await 할 때 실행이 시작됩니다. 그 시점에, 실행은 첫 번째 일드 표현식까지 진행한 후, 거기에서 다시 일시 중지(suspend)하고 await 중인 코루틴에게 *expression_list* 의 값을 돌려줍니다. 제너레이터에서처럼, 일시 중지된다는 것은, 모든 지역 상태가 보존된다는 뜻인데, 지역 변수들의 현재 연결들, 명령 포인터(instruction pointer), 내부 연산 스택(internal evaluation stack), 모든 예외 처리 상태가 포함됩니다. 비동기 제너레이터의 메서드가 돌려준 다음 객체를 await 해서 실행이 재개될 때, 함수는 마치 일드 표현식이 단지 또 하나의 외부 호출인 것처럼 진행할 수 있습니다. 재개된 후에 일드 표현식의 값은 실행을 재개하도록 만든 메서드에 달려있습니다. *__anext__()* 가 사용되었다면 결과는 None 입니다. 그렇지 않고, *asend()* 가 사용되었다면, 결과는 그 메서드로 전달된 값입니다.

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.

파이널리제이션을 처리하기 위해, 이벤트 루프는 파이널라이저 *(finalizer)* 함수를 정의해야 하는데 비동기 제너레이터-이터레이터를 받아서 아마도 *aclose()* 를 호출하고 그 코루틴을 실행합니다. 이 파이널라이저 는 sys.set_asyncgen_hooks() 을 호출해서 등록할 수 있습니다. 처음 탐색 될 때, 비동기 제너레이터-이터레이터는 파이널리제이션때 호출될 등록된 파이널라이저 를 저장할 것입니다. 파이널라이저

메서드의 참조할만한 예는 Lib/asyncio/base_events.py 에 있는 asyncio.Loop.shutdown_asyncgens 구현을 보세요.

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.

coroutine agen.**__anext__**()
어웨이터블을 돌려주는데, 실행하면 비동기 제너레이터 함수의 실행을 시작하거나 마지막으로 실행된 일드 표현식에서 재개합니다. 비동기 제너레이터 함수가 *__anext__()* 메서드로 재개될 때, 반환된 어웨이터블에서 현재의 일드 표현식은 항상 None 값을 갖고 반환된 어웨이터블을 실행하면 다음 일드 표현식까지 이어집니다. 일드 표현식의 *expression_list* 의 값은 종료하는 코루틴이 일으킨 StopIteration 의 값입니다. 비동기 제너레이터가 다른 값을 yield 하지 않고 종료되면, 비동기 탐색의 종료를 알리기 위해 어웨이터블이 대신 StopAsyncIteration 예외를 일으킵니다.

이 메서드는 보통 *async for* 루프에 의해 묵시적으로 호출됩니다.

coroutine agen.**asend**(*value*)
어웨이터블을 돌려주는데, 실행하면 비동기 제너레이터의 실행을 재개합니다. 제너레이터의 *send()* 메서드 처럼, 이것은 값을 비동기 제너레이터 함수로 "보내(send)"고, *value* 인자는 현재 일드 표현식의 결과가 됩니다. *asend()* 메서드가 돌려주는 어웨이터블은 제너레이터가 yield 하는 다음 값을 발생시킨 StopIteration 의 값으로 돌려주거나, 비동기 제너레이터가 다른 값을 yield 하지 않고 종료하면 StopAsyncIteration 를 일으킵니다. 비동기 제너레이터를 시작시키도록 *asend()* 가 호출될 때, 값을 받을 일드 표현식이 없으므로 인자를 None 으로 호출해야 합니다.

coroutine 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.

coroutine 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
```

프라이머리는 값을 구했을 때 어트리뷰트 참조를 지원하는 형의 객체가 나와야 하는데, 대부분 객체가 이 상황에 해당합니다. 이 객체는 그 이름을 식별자로 하는 어트리뷰트를 생산하도록 요청받습니다. 이 생산은 *__getattr__()* 메서드를 재정의해서 커스터마이즈 할 수 있습니다. 이 어트리뷰트가 없으면, `AttributeError` 을 일으킵니다. 그렇지 않으면, 생산된 객체의 형과 값은 그 객체에 의해 결정됩니다. 같은 어트리뷰트 참조의 값을 여러 번 구하면 각기 다른 객체가 얻어질 수 있습니다.

6.3.2 서브스크립션(Subscriptions)

서브스크립션은 시퀀스(문자열, 튜플, 리스트)나 매핑(딕셔너리) 객체의 항목을 선택합니다:

```
subscription  ::=  primary "[" expression_list "]"
```

프라이머리는 값을 구했을 때 서브스크립션을 지원하는 객체가 나와야 합니다(예를 들어, 리스트나 딕셔너리). 사용자 정의 객체들은 *__getitem__()* 메서드를 구현해서 서브스크립션을 지원할 수 있습니다.

내장 객체들의 경우, 서브스크립션을 지원하는 두 가지 종류의 객체들이 있습니다:

프라이머리가 매핑이면, 표현식 목록은 값을 구했을 때 매핑의 키 중 하나가 되어야 하고, 서브스크립션은 매핑에서 그 키에 대응하는 값을 선택합니다. (표현식 목록은 정확히 하나의 항목을 가지는 경우만을 제외하고는 튜플입니다.)

프라이머리가 시퀀스면, 표현식 목록은 값을 구했을 때 정수나 슬라이스(slice)(다음 섹션에서 논의합니다)가 나와야 합니다.

형식 문법은 시퀀스에서 음수 인덱스에 대해 특별히 규정하지 않습니다; 하지만, 내장 시퀀스들은 모두 인덱스에 시퀀스의 길이를 더하는 것으로 음의 인덱스를 해석하는 *__getitem__()* 메서드를 제공합니다 (그래서 `x[-1]` 은 `x` 의 마지막 항목을 선택합니다). 결괏값은 반드시 시퀀스에 있는 항목들의 개수보다 작은 음이 아닌 정수가 되어야 하고, 서브스크립션은 인덱스가 그 값이 되는 항목을 선택합니다 (0에서부 터 센다). 음의 인덱스와 슬라이싱에 대한 지원이 객체의 *__getitem__()* 메서드에서 이루어지기 때문에, 이 메서드를 재정의하는 서브 클래스는 그 지원을 명시적으로 추가할 필요가 있습니다.

문자열의 항목은 문자입니다. 문자는 별도의 데이터형이 아니고, 하나의 문자만을 가진 문자열입니다.

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).

슬라이싱의 의미는 다음과 같습니다. 프라이머리가 (일반 서브스크립션과 같은 *__getitem__()* 메서드를 사용해서) 다음과 같이 슬라이스 목록으로부터 만들어지는 키로 인덱싱됩니다. 슬라이스 목록이 적어도 하나의 쉼표를 갖는다면, 키는 슬라이스 항목들의 변환을 포함하는 튜플입니다; 그렇지 않으면 슬라이스 항목 하나의 변환이 키입니다. 표현식인 슬라이스 항목의 변환은 그 표현식입니다. 고유한 슬라이스 (proper slice) 의 변환은 슬라이스 객체 (섹션 표준형 계층 를 보라) 인데, `start`, `stop`, `step` 어트리뷰트가 각각 `lower_bound`, `upper_bound`, `stride` 로 주어진 표현식이고, 빠진 표현식들을 `None` 으로 채웁니다.

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 ::=  ["*"] 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.

프라이머리의 값을 구하면 콜러블 객체 (사용자 정의 함수, 내장 함수, 내장 객체들의 메서드, 클래스 객체, 클래스 인스턴스의 메서드, *__call__()* 메서드를 갖는 모든 객체가 콜러블입니다) 가 나와야 합니다. 모든 인자 표현식들은 호출을 시도하기 전에 값이 구해집니다. 형식 매개변수 목록의 문법은 함수 정의 섹션을 참고하면 됩니다.

키워드 인자가 있으면, 먼저 다음과 같이 위치 인자로 변환됩니다. 먼저 형식 매개변수들의 채워지지 않은 슬롯들의 목록이 만들어집니다. N 개의 위치 인자들이 있다면, 처음 N 개의 슬롯에 넣습니다. 그다음, 각 키워드 인자마다, 식별자가 대응하는 슬롯을 결정하는 데 사용됩니다 (식별자가 첫 번째 형식 매개변수의 이름과 같으면, 첫 번째 슬롯은 사용되고, 이런 식으로 계속합니다). 슬롯이 이미 채워졌으면, `TypeError` 예외를 일으킵니다. 그렇지 않으면 그 인자의 값을 슬롯에 채워 넣습니다 (표현식이 `None` 이라 할지라도, 슬롯을 채우게 됩니다). 모든 인자가 처리되었을 때, 아직 채워지지 않은 슬롯들을 함수 정의로부터 오는 대응하는 기본값들로 채웁니다. (기본값들은 함수가 정의될 때 한 번만 값을 구합니다; 그래서, 리스트나 딕셔너리 같은 가변객체들이 기본값으로 사용되면 해당 슬롯에 인자값을 지정하지 않은 모든 호출에서 공유됩니다; 보통 이런 상황은 피해야 할 일입니다.) 만약 기본값이 지정되지 않고, 아직도 비어있는 슬롯이 남아있다면, `TypeError` 예외가 발생합니다. 그렇지 않으면, 채워진 슬롯의 목록이 호출의 인자 목록으로 사용됩니다.

CPython implementation detail: 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+4 개의 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
```

같은 호출에서 키워드 인자와 *expression 문법을 모두 사용하는 것은 일반적이지 않기 때문에, 실제로는 이런 혼란이 일어나지 않습니다.

문법 **expression 이 함수 호출에 등장하면, expression 의 값은 매핑 이 되어야 합니다, 그 내용이 추가의 키워드 인자인 것처럼 취급됩니다. 키워드가 (명시적인 키워드 인자나 다른 언 패킹으로부터) 이미 존재한다면 TypeError 예외가 발생합니다.

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: 인자 목록을 전달해서 함수의 코드 블록이 실행됩니다. 코드 블록이 처음으로 하는 일은 형식 매개변수들을 인자에 결합하는 것입니다; 이것은 섹션 함수 정의 에서 설명합니다. 코드 블록이 *return* 문을 실행하면, 함수 호출의 반환 값을 지정하게 됩니다.

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: 그 클래스는 *__call__()* 메서드를 정의해야 합니다; 그 효과는 그 메서드가 호출되는 것과 같습니다.

6.4 어웨이트 표현식

어웨이터블 에서 코루틴 의 실행을 일시 중지합니다. 오직 코루틴 함수 에서만 사용할 수 있습니다.

```
await_expr   ::=   "await" primary
```

버전 3.5에 추가.

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.)

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
```

일 항 `-` (마이너스) 연산자는 그 숫자 인자의 음의 값을 줍니다.

일 항 `+` (플러스) 연산자는 그 숫자 인자의 값을 변경 없이 줍니다.

일 항 `~` (인버트) 연산자는 그 정수 인자의 비트 반전된 값을 줍니다. `x` 의 비트 반전은 `-(x+1)` 로 정의됩니다. 오직 정수에만 적용됩니다.

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
```

`*` (곱셈) 연산자는 인자들의 곱을 줍니다. 인자들은 모두 숫자거나, 한 인자는 정수고 다른 인자는 시퀀스여야 합니다. 앞의 경우에, 숫자들은 공통 형으로 변환된 후 곱해집니다. 후자의 경우, 시퀀스의 반복이 수행됩니다; 음의 반복 값은 빈 시퀀스를 만듭니다.

`@` (at) 연산자는 행렬 곱셈에 사용하려는 것입니다. 파이썬의 내장형들 어느 것도 이 연산자를 구현하지 않습니다.

버전 3.5에 추가.

/ (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.

`%` (모듈로, modulo) 연산자는 첫 번째 인자를 두 번째 인자로 나눈 나머지를 줍니다. 숫자 인자들은 먼저 공통형으로 변환됩니다. 오른쪽 인자가 0이면 `ZeroDivisionError` 예외를 일으킵니다. 인자들은 실수가 될 수 있습니다, 예를 들어, `3.14%0.7` 는 `0.34` 와 같습니다 (`3.14` 가 `4*0.7 + 0.34` 와 같으므로.) 모듈로 연산자는 항상 두 번째 피연산자와 같은 부호를 갖는 결과를 줍니다 (또는 0입니다); 결과의 절댓값은 두 번째 피연산자의 절댓값보다 작습니다[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.

정수 나눗셈 연산자, 모듈로 연산자, `divmod()` 함수는 복소수에 대해서는 정의되어 있지 않습니다. 대신, 적절하다면, `abs()` 함수를 사용해서 실수로 변환하십시오.

+ (덧셈) 연산자는 그 인자들의 합을 줍니다. 인자들은 둘 다 숫자거나, 둘 다 같은 형의 시퀀스여야 합니다. 앞의 경우, 숫자들은 먼저 공통형으로 변환된 후, 함께 합쳐집니다. 후자의 경우 시퀀스는 이어붙이게 됩니다.

- (빼기) 연산자는 그 인자들의 차를 줍니다. 숫자 인자들은 먼저 공통형으로 변환됩니다.

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).

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
```

`&` 연산자는 그 인자들의 비트별 AND를 주는데, 인자들은 정수여야 합니다.

[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.

^ 연산자는 그 인자들의 비트별 XOR (배타적 OR)를 주는데, 인자들은 정수여야 합니다.

| 연산자는 그 인자들의 비트별 (포함적, inclusive) OR를 주는데, 인자들은 정수여야 합니다.

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"
```

비교는 논리값을 줍니다: `True` 또는 `False`

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.

모든 형은 (직접적 혹은 간접적으로) `object` 의 서브 형이기 때문에, 그들은 `object` 로 부터 기본 비교 동작을 계승합니다. 형들은 *`__lt__()`* 와 같은 풍부한 비교 메서드(*rich comparison methods*) 를 구현해서 자신의 비교 동작을 커스터마이즈할 수 있는데, 기본적인 커스터마이제이션 에서 설명됩니다.

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.

The not-a-number values `float('NaN')` and `decimal.Decimal('NaN')` are special. Any ordered comparison of a number to a not-a-number value is false. A counter-intuitive implication is that not-a-number values are not equal to themselves. For example, if `x = float('NaN')`, `3 < x`, `x < 3` and `x == x` are all false, while `x != x` is true. This behavior is compliant with IEEE 754.

- 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.

  시퀀스는 대응하는 요소 간의 비교를 사용해서 사전적으로 비교하는데, 요소들의 반사성(reflexivity)이 강제됩니다.

  요소들의 반사성을 강제한다는 것은, 컬렉션의 비교가 컬렉션 요소 `x` 에 대해, `x == x` 가 항상 참이라고 가정한다는 것입니다. 그 가정에 기반을 둬서, 요소들의 아이덴티티가 먼저 비교된 후에, 이것이 다를 때만 요소 간의 동등 비교가 수행됩니다. 비교되는 요소들이 반사적일 때, 이런 접근법은 엄밀한 요소 간 비교와 같은 결과를 줍니다. 비 반사적인 요소의 경우, 결과가 엄밀한 요소 비교와 달라질 수 있고, 놀랄 수 있습니다: 예를 들어, 비 반사적인 NaN이 리스트에서 사용될 때 다음과 같은 비교 동작을 보입니다:

```
>>> nan = float('NaN')
>>> nan is nan
True
>>> nan == nan
False                  <-- the defined non-reflexive behavior of NaN
>>> [nan] == [nan]
True                   <-- list enforces reflexivity and tests identity first
```

  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).
- 매핑들 (`dict` 의 인스턴스들) 은 같은 *(key, value)* 쌍들을 가질 때, 그리고 오직 이 경우만 같다고 비교됩니다. 키와 값의 동등 비교는 반사성을 강제합니다.

  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,

[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.

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 갖거나 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.

__contains__() 메서드를 정의하는 사용자 정의 클래스의 경우, `x in y` 는 `y.__contains__(x)` 가 참을 줄 때 `True` 를, 그렇지 않으면 `False` 를 돌려줍니다.

__contains__() 를 정의하지 않지만 *__iter__()* 를 정의하는 사용자 정의 클래스의 경우, `y` 를 이터레이트할 때 표현식 `x is z or x == z` 를 참이 되게 하는 어떤 값 `z` 가 만들어지면 `x in y` 는 `True` 입니다. 탐색하는 동안 예외가 발생하면 *in* 이 그 예외를 일으킨 것으로 취급됩니다.

마지막으로, 올드스타일(old-style) 이터레이션 프로토콜을 시도합니다: 클래스가 *__getitem__()* 를 정의하면, `x in y` 는 `x is y[i] or x == y[i]` 를 만족하는 음이 아닌 정수 인덱스 *i* 가 존재하고, 그보다 작은 모든 정수 인덱스들에 대해 `IndexError` 예외를 일으키지 않는 경우, 그리고 오직 그 경우만 `True` 가 됩니다. (그 밖의 예외가 발생하면 *in* 이 그 예외를 일으킨 것으로 취급됩니다.

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
```

논리 연산의 문맥에서, 그리고 표현식이 제어 흐름 문(control flow statements)에서 사용될 때, 다음 값들은 거짓으로 해석됩니다: `False`, `None`, 모든 형의 숫자 0, 빈 문자열과 컨테이너(문자열, 튜플, 리스트, 딕셔너리, 집합, 불변 집합(frozenset) 들을 포함합니다). 그 밖의 모든 값은 참으로 해석됩니다. 사용자 정의 객체들은 *__bool__()* 메서드를 제공해서 자신의 논리값(truth value)을 커스터마이즈 할 수 있습니다.

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.)

6.12 조건 표현식(Conditional expressions)

```
conditional_expression  ::=  or_test ["if" or_test "else" expression]
expression              ::=  conditional_expression | lambda_expr
expression_nocond       ::=  or_test | lambda_expr_nocond
```

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 *C* 의 값을 구합니다. *C* 가 참이면, *x* 의 값이 구해지고 그 값을 돌려줍니다; 그렇지 않으면, *y* 의 값을 구한 후에 그 결과를 돌려줍니다.

조건 표현식에 대한 더 자세한 내용은 **PEP 308** 를 참조하세요.

[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.

6.13 람다(Lambdas)

```
lambda_expr          ::=  "lambda" [parameter_list] ":" expression
lambda_expr_nocond   ::=  "lambda" [parameter_list] ":" expression_nocond
```

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.

6.14 표현식 목록(Expression lists)

```
expression_list      ::=  expression ("," expression)* [","]
starred_list         ::=  starred_item ("," starred_item)* [","]
starred_expression   ::=  expression | (starred_item ",")* [starred_item]
starred_item         ::=  expression | "*" or_expr
```

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.

버전 3.5에 추가: 표현식 목록에서의 이터러블 언 패킹, **PEP 448** 에서 최초로 제안되었습니다.

끝에 붙는 쉼표는 단일 튜플(single tuple) (소위, 싱글톤(*singleton*)) 을 만들 때만 필수입니다; 다른 모든 경우에는 생략할 수 있습니다. 끝에 붙는 쉼표가 없는 단일 표현식은 튜플을 만들지 않고, 그 표현식의 값을 줍니다. (빈 튜플을 만들려면, 빈 괄호 쌍을 사용하십시오: `()`.)

6.15 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
```

6.16 연산자 우선순위

다음 표는 파이썬 에서의 연산자 우선순위를 가장 낮은 것 (least binding) 에서 가장 높은 것 (most binding) 순으로 요약합니다. 같은 상자에 들어있는 연산자들은 같은 우선순위를 갖습니다. 문법이 명시적으로 주어지지 않는 이상, 연산자들은 이항 (binary) 입니다. 같은 상자에 있는 연산자들은 왼쪽에서 오른쪽으로 그룹 지어집니다 (거듭제곱은 예외인데, 오른쪽에서 왼쪽으로 그룹 지어집니다).

비교, 멤버십 검사, 아이덴티티 검사들은 모두 같은 우선순위를 갖고 비교 섹션에서 설명한 것처럼 왼쪽에서 오른쪽으로 이어붙이기 (chaining) 하는 기능을 갖습니다.

연산자	설명
lambda	람다 표현식
if – else	조건 표현식
or	논리 OR
and	논리 AND
not x	논리 NOT
in, *not in*, *is*, *is not*, <, <=, >, >=, !=, ==	비교, 멤버십 검사와 아이덴티티 검사를 포함합니다
|	비트 OR
^	비트 XOR
&	비트 AND
<<, >>	시프트
+, –	덧셈과 뺄셈
*, @, /, //, %	곱셈, 행렬 곱셈, 나눗셈, 정수 나눗셈, 나머지[5]
+x, –x, ~x	양, 음, 비트 NOT
**	거듭제곱[6]
await x	어웨이트 표현식
x[index], x[index:index], x(arguments...), x.attribute	서브스크립션, 슬라이싱, 호출, 어트리뷰트 참조
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or parenthesized expression, list display, dictionary display, set display

[5] % 연산자는 문자열 포매팅에도 사용됩니다; 같은 우선순위가 적용됩니다.
[6] 거듭제곱 연산자 ** 는 오른쪽에 오는 산술이나 비트 일 항 연산자보다 약하게 결합합니다, 즉, 2**-1 는 0.5 입니다.

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
```

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.
- 그렇지 않으면: 객체는 타깃 목록에 나오는 타깃의 수와 같은 수의 항목들을 제공하는 이터러블이어야 하고, 항목들은, 왼쪽에서 오른쪽으로, 대응하는 타깃들에 대입됩니다.
  - 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 객

체에 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, RHS 표현식, `a.x` 는 인스턴스 어트리뷰트나 (인스턴스 어트리뷰트가 없다면) 클래스 어트리뷰트를 액세스할 수 있습니다. LHS 타깃 `a.x` 는 항상 필요하면 만들어서라도 항상 인스턴스 어트리뷰트를 설정합니다. 그래서, 두 `a.x` 가 같은 어트리뷰트를 가리키는 것은 필요조건이 아닙니다: RHS 표현식이 클래스 어트리뷰트를 가리킨다면, LHS 는 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).

  프라이머리가 (딕셔너리 같은) 매핑 객체면, 서브 스크립트는 매핑의 키 형과 호환되는 형이어야 하고, 매핑에 그 서브 스크립트를 객체에 매핑하는 키/데이터 쌍을 만들도록 요청합니다. 이때 같은 킷값을 갖는 기존의 키/값 쌍을 대체할 수도 있고, (같은 값의 키가 존재하지 않는 경우) 새 키/값 쌍을 삽입할 수도 있습니다.

  사용자 정의 객체의 경우는, 적절한 인자로 *`__setitem__()`* 메서드가 호출됩니다.

- 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.

CPython implementation detail: 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 안 에서의 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.

`x += 1` 과 같은 증분 대입 표현은 `x = x + 1` 처럼 다시 쓸 수 있는데, 정확히 같은 효과는 아니지만 비슷한 결과를 줍니다. 증분 버전에서는, `x` 의 값을 오직 한 번만 구합니다. 또한, 가능할 때, 실제 연산은 제자리 *(in-place)* 에서 수행되는데, 새 객체를 만들고 그것을 타깃에 대입하기보다는, 예전 객체를 수정한다는 의미입니다.

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 ["=" expression]
```

일반 대입문 과의 차이점은 오직 하나의 타깃과 오직 하나의 우변 값만 허락된다는 것입니다.

대인 타깃에 단순한 이름을 쓰는 경우, 클래스나 모듈 스코프에 있으면, 어노테이션은 값이 구해진 후 특별한 클래스나 모듈의 어트리뷰트 `__annotations__` 에 저장되는데, 이 어트리뷰트는 (만약 비공개면 뒤섞인) 변수 이름을 어노테이션의 값으로 대응시키는 딕셔너리 매핑입니다. 이 어트리뷰트는 쓰기가 허락되고, 클래스나 모듈의 실행을 시작할 때 어노테이션이 정적으로 발견되면 만들어집니다.

대입 타깃으로 표현식을 쓸 때, 어노테이션은 클래스나 모듈 스코프에 있는 것처럼 값이 구해지지만 저장되지는 않습니다.

이름이 함수 스코프에서 어노테이트되면, 이 이름은 그 스코프에 지역적 (local) 입니다. 함수 스코프에서 어노테이션은 값이 구해지거나 저장되지 않습니다.

우변이 존재하면, 어노테이트된 대입은 (적절한 곳에서) 어노테이션의 값을 구하기 전에 실제 대입을 수행합니다. 표현식 타깃의 경우 우변이 존재하지 않으면, 인터프리터는 티깃의 값을 구하는데, 마지막 *`__setitem__()`* 이나 *`__setattr__()`* 호출은 생략합니다.

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.

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)
```

이 동등성 들은 `__debug__` 과 `AssertionError` 가 같은 이름의 내장 변수들을 가리킨다고 가정합니다. 현재 구현에서, 내장 변수 `__debug__` 은 일반적인 상황에서 `True` 이고, 최적화가 요청되었을 때 (명령행 옵션 `-O`) `False` 입니다. 현재의 코드 생성기는 컴파일 시점에 최적화가 요청되면 assert 문을 위한 코드를 만들지 않습니다. 에러 메시지에 실패한 표현식의 소스 코드를 포함할 필요가 없음에 주의하십시오; 그것은 스택 트레이스의 일부로 출력됩니다.

`__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
```

yield 문은 *yield* 표현식 과 같은 의미가 있습니다. 동등한 yield 표현식에서 필요로 하는 괄호를 생략하기 위해 yield 문을 사용합니다. 예를 들어, yield 문

```
yield <expr>
yield from <expr>
```

은 다음과 같은 yield 표현식 문장들과 동등합니다

```
(yield <expr>)
(yield from <expr>)
```

yield 표현식과 문장은 제너레이터 함수를 정의할 때만 사용되고, 제너레이터 함수의 바디에서만 사용됩니다. 함수 정의가 일반 함수 대신 제너레이터 함수를 만들도록 하는 데는 yield를 사용하는 것만으로 충분합니다.

yield 의 뜻에 대한 전체 세부 사항들은 일드 표현식 *(Yield expressions)* 섹션을 참고하면 됩니다.

7.8 raise 문

```
raise_stmt  ::=  "raise" [expression ["from" expression]]
```

표현식이 주어지지 않으면, *raise* 는 현재 스코프에서 활성화된 마지막 예외를 다시 일으킵니다. 현재 스코프에 활성화된 예외가 없다면, 이것이 에러라는 것을 알리기 위해 RuntimeError 예외를 일으킵니다.

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.

트레이스백 객체는 보통 예외가 일어날 때 자동으로 만들어지고 쓰기 가능한 __traceback__ 어트리뷰트로 첨부됩니다. 다음과 같이, with_traceback() 예외 메서드를 사용하면, 예외를 만들고 트레이

스백을 직접 설정하는 것을 한 번에 할 수 있습니다 (같은 예외 인스턴스를 돌려주는데, 그 인자값으로 트레이스백을 설정해줍니다):

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

`from` 절은 예외 연쇄 (exception chaining) 에 사용됩니다. 주어진다면, 두 번째 표현식 *(expression)* 은 또 하나의 예외 클래스나 인스턴스야 되는데, 발생한 예외에 (쓰기 가능한) `__cause__` 어트리뷰트로 첨부됩니다. 발생한 예외가 처리되지 않으면, 두 예외가 모두 인쇄됩니다:

```
>>> 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>
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>
RuntimeError: Something bad happened
```

예외 처리기나 *finally* 절에서 예외가 발생하면 비슷한 메커니즘이 묵시적으로 적용됩니다: 앞선 예외가 새 예외의 `__context__` 어트리뷰트로 첨부됩니다.

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

예외 연쇄는 `from` 절에 `None` 을 지정해서 명시적으로 중지시킬 수 있습니다:

```
>>> 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.

버전 3.3에 추가: 예외 문맥 (exception context) 의 자동 출력을 제한할 수 있는 `__suppress_context__` 어트리뷰트

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 정의 또는 그 루프 내의 *finally* 에 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" 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.

첫 번째 단계, 모듈을 찾고 로드하는 것의 세부 사항은 임포트 시스템 에 있는 섹션에서 아주 상세하게 설명하는데, 임포트될 수 있는 여러 종류의 패키지와 모듈들과 임포트 시스템을 커스터마이즈하는데 사용될 수 있는 모든 훅에 관해서도 설명하고 있습니다.

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.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo.bar.baz imported and bound as fbb
from foo.bar import baz    # foo.bar.baz imported and 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.

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

파이썬 3.7에서 퓨처 문을 사용해야 하는 유일한 기능은 `annotations` 입니다.

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.

퓨처 문을 포함하는 모듈 `M` 에 등장하는 내장 함수 `exec()` 와 `compile()` 를 호출해서 컴파일되는 코드는, 기본적으로는, 퓨처 문이 지정하는 새 문법과 개념을 사용합니다. 이것은 `compile()` 에 주는 생략 가능한 인자로 제어될 수 있습니다 — 자세한 내용은 그 함수의 문서를 보십시오.

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)*
```

global 문은 현재 코드 블록 전체에 적용되는 선언입니다. 나열된 식별자들이 전역으로 해석되어야 한다는 뜻입니다. global 선언 없이 자유 변수들이 전역을 가리킬 수 있기는 하지만, `global` 없이 전역 변수에 값을 대입하는 것은 불가능합니다.

global 문에 나열된 이름들은 같은 코드 블록에서 `global` 문 앞에 등장할 수 없습니다.

global 문에 나열된 이름들은 형식 매개변수나 *for* 루프 제어 타깃, 클래스(*class*) 정의, 함수 정의, 임포트(*import*) 문, 변수 어노테이션으로 정의되지 않아야 합니다.

CPython implementation detail: 현재 구현이 이 제약들의 일부를 강제하지 않지만, 프로그램은 이 자유를

남용하지 말아야 하는데, 미래의 구현은 그것들을 강제하거나 프로그램의 의미를 예고 없이 변경할 수 있기 때문입니다.

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)*
```

nonlocal 문은 나열된 식별자들이 전역을 제외하고 가장 가까이서 둘러싸는 스코프에서 이미 연결된 변수를 가리키도록 만듭니다. 이것은 중요한데, 연결의 기본 동작이 지역 이름 공간을 먼저 검색하는 것이기 때문입니다. 이 문장은 캡슐화된 코드가 전역 (모듈) 스코프 외에 지역 스코프 밖의 변수들에 재연결할 수 있도록 합니다.

nonlocal 문에 나열된 이름들은, *global* 문에 나열된 것들과는 달리, 둘러싼 스코프에서 이미 존재하는 연결들을 가리켜야만 합니다 (새 연결이 어떤 스코프에 만들어져야만 하는지 명확하게 결정할 수 없습니다).

nonlocal 문에 나열되는 이름들은 지역 스코프에 이미 존재하는 연결들과 겹치지 않아야 합니다.

더 보기:

PEP 3104 - 바깥 스코프에 있는 이름들에 대한 액세스 *nonlocal* 문의 규격.

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
                    | 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" expression ":" suite
              ("elif" 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" 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" expression_list ":" suite
               ["else" ":" suite]
```

표현식 목록은 한 번만 값이 구해집니다; 이터러블 객체가 나와야 합니다. expression_list 의 결과로 이터레이터가 만들어집니다. 그런 다음 이터레이터가 제공하는 항목마다, 이터레이터가 돌려주는 순서대로, 스위트가 한 번씩 실행됩니다. 순환마다 각 항목이 대입의 표준 규칙 (대입문 을 보세요) 으로 타깃 목록에 대입된 다음, 스위트가 실행됩니다. 항목들이 소진되었을 때 (이터레이터가 StopIteration 예외를 일으킬 때나 빈 시퀀스인 경우는 즉시 발생합니다), the else 절의 스위트가 (있다면) 실행되고 루프를 종료합니다.

첫 번째 스위트에서 실행되는 *break* 문은 else 절을 실행하지 않고 루프를 종료합니다. 첫 번째 스위트

에서 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
```

타깃 목록의 이름들은 루프가 종료될 때 삭제되지 않지만, 시퀀스가 비어있다면, 루프에 의해 전혀 대입이 일어나지 않을 수도 있습니다. 힌트: 내장 함수 `range()` 는 파스칼의 `for i := a to b do` 의 효과를 흉내 내는데 적합한 정수의 이터레이터를 돌려줍니다; 예를 들어, `list(range(3))` 는 리스트 `[0, 1, 2]` 를 돌려줍니다.

참고: 시퀀스가 루프에 의해 수정될 때는 미묘한 점이 있습니다 (이것은 오직 가변 시퀀스에서만 일어납니다, 가령 리스트). 다음에 어떤 항목이 사용될지를 추적하는 내부 카운터가 사용되고, 각 이터레이션마다 증가합니다. 이 카운터가 시퀀스의 길이에 도달하면 루프가 종료됩니다. 이것은 만약 스위트가 시퀀스에서 현재 (또는 그 이전의) 항목을 삭제하면, 다음 항목을 건너뛰게 된다는 뜻입니다 (다음 항목이 이미 다뤄진 현재 항목의 인덱스를 갖게 되기 때문입니다). 마찬가지로, 스위트가 현재 항목 앞으로 시퀀스에 항목을 삽입하면, 현재 항목은 루프의 다음 순환에서 현재 항목이 한 번 더 다뤄지게 됩니다. 이것은 고약한 버그로 이어질 수 있는데, 전체 시퀀스의 슬라이스로 임시 사본을 만듦으로써 피할 수 있습니다, 예를 들어

```
for x in a[:]:
    if x < 0: a.remove(x)
```

8.4 `try` 문

try 문은 문장 그룹에 대한 예외 처리기나 정리 (cleanup) 코드 또는 그 둘 모두를 지정하는 데 사용됩니다.

```
try_stmt   ::=  try1_stmt | try2_stmt
try1_stmt  ::=  "try" ":" suite
                ("except" [expression ["as" identifier]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try2_stmt  ::=  "try" ":" suite
                "finally" ":" suite
```

except 절 (들) 은 하나나 그 이상의 예외 처리기를 지정합니다. *try* 절에서 예외가 발생하지 않으면 아무런 예외 처리기도 실행되지 않습니다. `try` 스위트에서 예외가 발생할 때, 예외 처리기 검색이 시작됩니다. 이 검색은 그 예외에 매치되는 것을 발견할 때까지 except 절을 차례대로 들여다봅니다. 표현식이 없는 except 절이 있다면 가장 마지막에 와야 합니다; 모든 예외와 매치됩니다. 표현식이 있는 except 절의 경우, 표현식의 값을 구하고, 결과 객체가 예외와 "호환" 되면 그 절이 예외에 매치됩니다. 객체는 예외 객체의 클래스나 베이스 클래스일 때, 또는 예외와 호환되는 항목을 포함한 튜플일 때 예외와 호환됩니다.

except 절 중 어느 것도 예외와 매치되지 않으면, 예외 처리기 검색은 둘러싼 코드와 호출 스택에서 계속됩니다.[1]

만약 except 절의 헤더에 있는 표현식의 값을 구할 때 예외가 발생하면, 원래의 처리기 검색은 취소되고 둘러싼 코드와 호출 스택에서 새 예외에 대해 검사가 시작됩니다 (*try* 문 전체가 예외를 일으킨 것으로 취급됩니다).

[1] 다른 예외를 일으키는 *finally* 절이 있지 않은 한 예외는 호출 스택으로 퍼집니다. 그 새 예외는 예전의 것을 잃어버리게 만듭니다.

매치되는 except 절이 발견되면, 예외는 그 except 절에 있는 as 키워드(가 있다면) 뒤에 지정된 타깃에 대입되고, except 절의 스위트가 실행됩니다. 모든 except 절은 실행 가능한 블록을 가져야 합니다. 블록의 끝에 도달하면, try 문 전체의 뒤에서 일반적인 실행이 계속됩니다. (이것은 같은 예외에 대해 두 개의 중첩된 처리기가 있고, 예외가 안쪽 처리기의 try 절에서 발생했다면, 바깥 처리기는 예외를 처리하지 않게 된다는 뜻이 됩니다.)

예외가 as target 을 사용해서 대입될 때, except 절 끝에서 삭제됩니다. 이것은 마치

```
except E as N:
    foo
```

가 이렇게 변환되는 것과 같습니다

```
except E as N:
    try:
        foo
    finally:
        del N
```

이것은 except 절 후에 참조하려면 예외를 다른 이름에 대입해야 한다는 뜻입니다. 예외를 제거하는 이유는, 그것에 첨부된 트레이스백으로 인해, 스택 프레임과 참조 순환을 형성해서 다음 가비지 수거가 일어나기 전까지 그 프레임의 모든 지역 변수들을 잡아두기 때문입니다.

except 절의 스위트가 실행되기 전에, 예외의 상세 내용이 sys 모듈에 저장되는데, sys.exc_info() 를 통해 액세스할 수 있습니다. sys.exc_info() 는 예외 클래스, 예외 인스턴스, 예외가 프로그램의 어디에서 발생했는지를 알려주는 트레이스백 객체 (표준형 계층 를 보세요) 로 이루어진 3-튜플을 돌려줍니다. sys.exc_info() 값들은 예외를 처리한 함수로부터 복귀할 때 이전 값으로 복구됩니다.

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.

finally 가 있으면, '정리(cleanup)' 처리기를 지정합니다. *except* 와 else 절을 포함해서, 먼저 *try* 절이 실행됩니다. 이 절들의 어디에서건 예외가 일어나면, 예외는 임시 저장됩니다. finally 절이 실행됩니다. 만약 저장된 예외가 있으면, finally 절의 끝에서 다시 발생시킨다. finally 절이 다른 예외를 일으키면, 저장된 예외는 새 예외의 컨텍스트(context) 로 설정됩니다. finally 절이 *return* 이나 *break* 문을 실행하면, 저장된 예외는 버립니다.

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

finally 절을 실행하는 동안 예외 정보는 프로그램에 제공되지 않습니다.

try…finally 문의 *try* 스위트에서 *return*, *break*, *continue* 문이 실행될 때, *finally* 절도 '나가는 길에' 실행됩니다. finally 절에서는 *continue* 문을 사용할 수 없습니다. (그 이유는 현재 구현에 있는 문제 때문입니다 — 이 제약은 미래에 제거될 수 있습니다).

함수의 반환 값은 마지막에 실행된 *return* 문으로 결정됩니다. *finally* 절이 항상 실행되기 때문에, finally 절에서 실행되는 return 문이 항상 마지막에 실행되는 것이 됩니다:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> foo()
'finally'
```

예외에 관한 추가의 정보는 예외 섹션에서 찾을 수 있고, 예외를 일으키기 위해 *raise* 문을 사용하는 것에 관한 정보는 *raise* 문 섹션에서 찾을 수 있습니다.

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_item (","  with_item)* ":" suite
with_item   ::=   expression ["as" target]
```

하나의 "item" 을 사용하는 *with* 문의 실행은 다음과 같이 진행됩니다:

1. 컨텍스트 관리자를 얻기 위해 컨텍스트 표현식 (*with_item* 에 주어진 expression) 의 값을 구합니다.
2. 나중에 사용하기 위해 컨텍스트 관리자의 *__exit__()* 가 로드됩니다.
3. 컨텍스트 관리자의 *__enter__()* 메서드를 호출합니다.
4. *with* 문에 타깃이 포함되었으면, 그것에 *__enter__()* 의 반환 값을 대입합니다.

   참고: *with* 문은 *__enter__()* 메서드가 에러 없이 돌아왔을 때, *__exit__()* 가 항상 호출됨을 보장합니다. 그래서, 타깃에 대입하는 동안 에러가 발생하면, 스위트 안에서 에러가 발생한 것과 같이 취급됩니다. 아래의 6단계를 보세요.

   ---

5. 스위트가 실행됩니다.
6. 컨텍스트 관리자의 *__exit__()* 메서드를 호출합니다. 예외가 스위트를 종료되도록 만들었다면, 그것의 형, 값, 트레이스백이 *__exit__()* 의 인자로 전달됩니다. 그렇지 않으면 세 개의 None 이 인자로 공급됩니다.

   스위트가 예외 때문에 종료되었고, *__exit__()* 메서드의 반환 값이 거짓이면, 그 예외를 다시 일으킨다. 반환 값이 참이면, 예외를 억누르고, *with* 문 뒤에 오는 문장으로 실행을 계속합니다.

   스위트가 예외 이외의 이유로 종료되면, *__exit__()* 의 반환 값은 무시되고, 해당 종료의 종류에 맞는 위치에서 실행을 계속합니다.

하나 보다 많은 항목을 주면, 컨텍스트 관리자는 *with* 문이 중첩된 것처럼 진행합니다:

```
with A() as a, B() as b:
    suite
```

는 다음과 동등합니다

```
with A() as a:
    with B() as b:
        suite
```

버전 3.1에서 변경: 다중 컨텍스트 표현식의 지원

더 보기:

PEP 343 - "with" 문 파이썬 *with* 문의 규격, 배경, 예.

8.6 함수 정의

함수 정의는 사용자 정의 함수 객체 (표준형 계층 섹션을 보세요) 를 정의합니다:

```
funcdef                 ::=  [decorators] "def" funcname "(" [parameter_list] ")"
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
dotted_name             ::=  identifier ("." identifier)*
parameter_list          ::=  defparameter ("," defparameter)* ["," [parameter_list_sta
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" parameter
                             | "**" parameter [","]
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.[2]

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

```
@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.

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.

함수 정의가 실행될 때, 기본 매개변수 값은 왼쪽에서 오른쪽으로 값이 구해집니다. 이것은 표현식이 함수가 정의될 때 한 번 값이 구해지고, 호출마다 같은 "미리 계산된" 값이 사용된다는 것을 뜻합니다. 이것을 이해하는 것은 특히 기본값이 리스트나 딕셔너리와 같은 가변 객체일 때 중요합니다: 만약 함수가 그 객체를 수정하면 (가령, 리스트에 항목을 추가합니다), 그 결과 기본값이 수정됩니다. 이것은 일반적으로 의도하고 있는 것이 아니다. 이 문제를 회피하는 방법은 기본값으로 `None` 을 사용하고, 함수 바디에서 명시적으로 검사하는 것입니다, 예를 들어:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

함수 호출의 의미는 섹션 호출 에서 더 자세히 설명됩니다. 함수 호출은 항상 매개변수 목록에서 언급하는 모든 매개변수에 값을 대입하는데, 위치 인자들에서 올 수도, 키워드 인자들에서 올 수도, 기본값에서 올

[2] 함수 바디의 첫 번째 문장으로 등장하는 문자열 리터럴은 함수의 `__doc__` 어트리뷰트로 변환되어 함수의 독스트링 이 됩니다.

수도 있습니다. "*identifier" 형태가 존재하면, 남는 위치 매개변수들을 받는 튜플로 초기화됩니다. 기본값은 빈 튜플입니다. "**identifier" 형태가 존재하면, 남는 키워드 인자들을 받는 순서 있는 매핑으로 초기화됩니다. 기본값은 빈 매핑입니다. "*" 나 "*identifier" 뒤에 오는 매개변수들은 키워드 전용 매개변수들이고, 키워드 인자로만 전달될 수 있습니다.

매개변수들은 매개변수 이름 뒤에 오는 ": expression" 형태의 어노테이션을 가질 수 있습니다. 모든 매개변수는 어노테이션을 가질 수 있는데, *identifier 나 **identifier 형태조차 그렇습니다. 함수는 매개변수 목록 뒤에 오는 "-> expression" 형태의 반환("return") 어노테이션을 가질 수 있습니다. 이 어노테이션들은 올바른 파이썬 표현식이면 어떤 것이건 될 수 있습니다. 어노테이션의 존재는 함수의 의미를 바꾸지 않습니다. 어노테이션 값들은 함수 객체의 __annotations__ 어트리뷰트에서 매개변수의 이름을 키로 하는 딕셔너리의 값으로 제공됩니다. __future__ 에서 annotations 을 임포트하면, 지연된 평가가 활성화되어 어노테이션은 실행시간에 문자열로 보존됩니다. 그렇지 않으면 함수 정의가 실행될 때 평가됩니다. 이 경우 어노테이션은 소스 코드에 나오는 순서와 다른 순서로 평가될 수 있습니다.

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 표현식으로 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 클래스 변수 및 인스턴스 변수를 포함하는 변수 선언에 형 힌트를 줄 수 있는 기능

PEP 563 - 어노테이션의 지연된 평가 즉시 평가하는 대신 실행시간에 어노테이션을 문자열 형식으로 보존하여 어노테이션 내에서의 전방 참조를 지원합니다.

8.7 클래스 정의

클래스 정의는 클래스 객체(표준형 계층 섹션을 보세요)를 정의합니다:

```
classdef    ::=  [decorators] "class" classname [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 서브클래싱을 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.[3] 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.

클래스 바디에서 어트리뷰트가 정의되는 순서는, 새 클래스의 `__dict__` 에 보존됩니다. 이것은 클래스가 만들어진 직후에, 정의 문법을 사용해서 정의되는 클래스들에서만 신뢰할 수 있다는 것에 주의해야 합니다.

클래스 생성은 메타 클래스 를 사용해서 심하게 커스터마이즈할 수 있습니다.

클래스 역시 함수를 데코레이팅할 때처럼 데코레이트할 수 있습니다,

```
@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.

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.

8.8 코루틴

버전 3.5에 추가.

8.8.1 코루틴 함수 정의

```
async_funcdef ::= [decorators] "async" "def" funcname "(" [parameter_list] ")"
                  ["->" expression] ":" suite
```

파이썬 코루틴의 실행은 여러 지점에서 일시 중지되거나 재개될 수 있습니다(코루틴 을 보세요.). 코루틴의 바디 안에서, `await` 와 `async` 식별자는 예약 키워드가 됩니다; 어웨이트(*await*) 표현식, *async for*, *async with* 는 코루틴 바디에서만 사용할 수 있습니다.

`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:

[3] 클래스 바디의 첫 번째 문장으로 등장하는 문자열 리터럴은 그 이름 공간의 `__doc__` 항목으로 변환되어 클래스의 독스트링 이 됩니다.

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

8.8.2 async for 문

```
async_for_stmt  ::=  "async" for_stmt
```

비동기 이터러블 은 *iter* 구현에서 비동기 코드를 호출할 수 있고, 비동기 이터레이터 는 *next* 메서드에서 비동기 코드를 호출할 수 있습니다.

`async for` 문은 비동기 이터레이터에 대한 편리한 이터레이션을 허락합니다.

다음과 같은 코드는:

```
async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
```

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:
        BLOCK
else:
    BLOCK2
```

더 자세한 내용은 *__aiter__()* 를 *__anext__()* 보면 됩니다.

코루틴 함수의 바디 밖에서 `async for` 문을 사용하는 것은 `SyntaxError` 입니다.

8.8.3 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 EXPR as VAR:
    BLOCK
```

의미상으로 다음과 동등합니다:

```
mgr = (EXPR)
aexit = type(mgr).__aexit__
aenter = type(mgr).__aenter__(mgr)

VAR = await aenter
try:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    BLOCK
except:
    if not await aexit(mgr, *sys.exc_info()):
        raise
else:
    await aexit(mgr, None, None, None)
```

더 자세한 내용은 *__aenter__()* 와 *__aexit__()* 를 보면 됩니다.

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.

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)*
```

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
```

(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*
```

CHAPTER 10

전체 문법 규격

이것이 파서 제너레이터가 읽고, 파이썬 소스 파일을 파싱하는데 사용되는 전체 파이썬 문법 규칙입니다:

```
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: 'async' funcdef
funcdef: 'def' NAME parameters ['->' test] ':' suite

parameters: '(' [typedargslist] ')'
typedargslist: (tfpdef ['=' test] (',' tfpdef ['=' test])* [',' [
        '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',']]]
      | '**' tfpdef [',']]]
  | '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',']]]
  | '**' tfpdef [','])
tfpdef: NAME [':' test]
varargslist: (vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']
)
vfpdef: NAME

stmt: simple_stmt | compound_stmt
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist_star_expr))*)
annassign: ':' test ['=' test]
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal and annotated assignments, additional restrictions enforced by the␣
↪interpreter
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
             'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated | async_stmt
async_stmt: 'async' (funcdef | with_stmt | for_stmt)
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

test: or_test ['if' or_test 'else' test] | lambdef
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
# <> isn't actually a valid comparison operator in Python. It's here for the
# sake of a __future__ import described in PEP 401 (which really works :-)
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'@'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
atom_expr: ['await'] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [testlist_comp] ']' |
       '{' [dictorsetmaker] '}' |
       NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (test|star_expr) ( comp_for | (',' (test|star_expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( ((test ':' test | '**' expr)
                   (comp_for | (',' (test ':' test | '**' expr))* [','])) |
                  ((test | star_expr)
                   (comp_for | (',' (test | star_expr))* [',']))  )

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)*  [',']

# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
# "test '=' test" is really "keyword '=' test", but we have no such token.
# These need to be in a single rule to avoid grammar that is ambiguous
# to our LL(1) parser. Even though 'test' includes '*expr' in star_expr,
# we explicitly match '*' here, too, to give it proper precedence.
# Illegal combinations and orderings are blocked in ast.c:
# multiple (test comp_for) arguments are blocked; keyword unpackings
# that precede iterable unpackings are blocked; etc.
argument: ( test [comp_for] |
            test '=' test |
            '**' test |
            '*' test )

comp_iter: comp_for | comp_if
sync_comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_for: ['async'] sync_comp_for
comp_if: 'if' test_nocond [comp_iter]

# not used in grammar, but may appear in "node" passed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist
```

APPENDIX A

용어집

>>> 대화형 셸의 기본 파이썬 프롬프트. 인터프리터에서 대화형으로 실행될 수 있는 코드 예에서 자주 볼 수 있습니다.

... 들여쓰기 된 코드 블록의 코드를 입력할 때, 쌍을 이루는 구분자 (괄호, 대괄호, 중괄호) 안에 코드를 입력할 때, 데코레이터 지정 후의 대화형 셸의 기본 파이썬 프롬프트.

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

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

abstract base class (추상 베이스 클래스) 추상 베이스 클래스는 `hasattr()` 같은 다른 테크닉들이 불편하거나 미묘하게 잘못된 (예를 들어, 매직 메서드) 경우, 인터페이스를 정의하는 방법을 제공함으로써 덕 타이핑 을 보완합니다. ABC는 가상 서브 클래스를 도입하는데, 클래스를 계승하지 않으면서도 `isinstance()` 와 `issubclass()` 에 의해 감지될 수 있는 클래스들입니다; `abc` 모듈 설명서를 보세요. 파이썬에는 많은 내장 ABC 들이 따라오는데 다음과 같은 것들이 있습니다: 자료 구조 (`collections.abc` 모듈에서), 숫자 (`numbers` 모듈에서), 스트림 (`io` 모듈에서), 임포트 파인더와 로더 (`importlib.abc` 모듈에서). `abc` 모듈을 사용해서 자신만의 ABC를 만들 수도 있습니다.

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

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

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

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

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

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

- 위치 인자 *(positional argument)*: 키워드 인자가 아닌 인자. 위치 인자들은 인자 목록의 처음에 나오거나 이터러블 의 앞에 `*` 를 붙여 전달할 수 있습니다. 예를 들어, 다음과 같은 호출에서 `3` 과 `5` 는 모두 위치 인자입니다.

```
complex(3, 5)
complex(*(3, 5))
```

인자는 함수 바디의 이름 붙은 지역 변수에 대입됩니다. 이 대입에 적용되는 규칙들에 대해서는 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, 그 위치의 (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 이터레이터 가 *__anext__()* 가 돌려주는 또 하나의 어웨이터블로 재개되면, 떠난 곳으로 복귀합니다. **PEP 492**와 **PEP 525**를 보세요.

asynchronous iterable (비동기 이터러블) *async for* 문에서 사용될 수 있는 객체. *__aiter__()* 메서드는 비동기 이터레이터 를 돌려줘야 합니다. **PEP 492** 로 도입되었습니다.

asynchronous iterator (비동기 이터레이터) *__aiter__()* 와 *__anext__()* 메서드를 구현하는 객체. __anext__ 는 어웨이터블 객체를 돌려줘야 합니다. *async for*는 StopAsyncIteration 예외가 발생할 때까지 비동기 이터레이터의 *__anext__()* 메서드가 돌려주는 어웨이터블을 풉니다. **PEP 492**로 도입되었습니다.

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

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

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

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

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

bytes-like object (바이트열류 객체) bufferobjects 를 지원하고 C-연속 버퍼를 익스포트 할 수 있습니다. 여러 공통 `memoryview` 객체들은 물론이고 `bytes`, `bytearray`, `array.array` 객체들을 포함합니다. 바이트열류 객체들은 바이너리 데이터를 다루는 여러 가지 연산들에 사용될 수 있습니다; 압축, 바이너리 파일로 저장, 소켓을 통한 전송 같은 것들이 있습니다.

어떤 연산들은 바이너리 데이터가 가변적일 필요가 있습니다. 이런 경우에 설명서는 종종 "읽고-쓰기 바이트열류 객체"라고 표현합니다. 가변 버퍼 객체의 예로는 `bytearray` 와 `bytearray` 의 `memoryview` 가 있습니다. 다른 연산들은 바이너리 데이터가 불변 객체 ("읽기 전용 바이트열류 객체")에 저장되도록 요구합니다; 이런 것들의 예로는 `bytes`와 `bytes` 객체의 `memoryview` 가 있습니다.

bytecode (바이트 코드) 파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현입니다. 바이트 코드는 `.pyc` 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만듭니다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있습니다). 이 "중간 언어" 는

각 바이트 코드에 대응하는 기계를 실행하는 가상 기계 에서 실행된다고 말합니다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 합니다.

바이트 코드 명령어들의 목록은 dis 모듈 설명서에 나옵니다.

class (클래스) 사용자 정의 객체들을 만들기 위한 주형. 클래스 정의는 보통 클래스의 인스턴스를 대상으로 연산하는 메서드 정의들을 포함합니다.

class variable (클래스 변수) 클래스에서 정의되고 클래스 수준 (즉, 클래스의 인스턴스에서가 아니라) 에서만 수정되는 변수.

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

complex number (복소수) 익숙한 실수 시스템의 확장인데, 모든 숫자가 실수부와 허수부의 합으로 표현됩니다. 허수부는 실수에 허수 단위 (`-1`의 제곱근)를 곱한 것인데, 종종 수학에서는 `i`로, 공학에서는 `j`로 표기합니다. 파이썬은 후자의 표기법을 쓰는 복소수를 기본 지원합니다; 허수부는 `j` 접미사를 붙여서 표기합니다, 예를 들어, `3+1j`. `math` 모듈의 복소수 버전이 필요하면, `cmath`를 사용합니다. 복소수의 활용은 꽤 수준 높은 수학적 기능입니다. 필요하다고 느끼지 못한다면, 거의 확실히 무시해도 좋습니다.

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

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

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

coroutine (코루틴) Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the *`async def`* statement. See also **PEP 492**.

coroutine function (코루틴 함수) 코루틴 객체를 돌려주는 함수. 코루틴 함수는 *`async def`* 문으로 정의될 수 있고, *`await`* 와 *`async for`* 와 *`async with`* 키워드를 포함할 수 있습니다. 이것들은 **PEP 492** 에 의해 도입되었습니다.

CPython 파이썬 프로그래밍 언어의 규범적인 구현인데, python.org에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 "CPython" 이 사용됩니다.

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

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

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

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

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 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, 슈퍼클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 디스크립터 구현하기에 나옵니다.

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

dictionary view (딕셔너리 뷰) `dict.keys()`, `dict.values()`, `dict.items()` 메서드가 돌려주는 객체들을 딕셔너리 뷰라고 부릅니다. 이것들은 딕셔너리 항목들에 대한 동적인 뷰를 제공하는데, 딕셔너리가 변경될 때, 뷰가 이 변화를 반영한다는 뜻입니다. 딕셔너리 뷰를 완전한 리스트로 바꾸려면 `list(dictview)`를 사용하면 됩니다. dict-views를 보세요.

docstring (독스트링) 클래스, 함수, 모듈에서 첫 번째 표현식으로 나타나는 문자열 리터럴. 스위트가 실행될 때는 무시되지만, 컴파일러에 의해 인지되어 둘러싼 클래스, 함수, 모듈의 `__doc__` 어트리뷰트로 삽입됩니다. 인트로스팩션을 통해 사용할 수 있으므로, 객체의 설명서를 위한 규범적인 장소입니다.

duck-typing (덕 타이핑) 올바른 인터페이스를 가졌는지 판단하는데 객체의 형을 보지 않는 프로그래밍 스타일; 대신, 단순히 메서드나 어트리뷰트가 호출되거나 사용됩니다 ("오리처럼 보이고 오리처럼 꽥꽥댄다면, 그것은 오리다.") 특정한 형 대신에 인터페이스를 강조함으로써, 잘 설계된 코드는 다형적인 치환을 허락함으로써 유연성을 개선할 수 있습니다. 덕 타이핑은 `type()` 이나 `isinstance()`을 사용한 검사를 피합니다. (하지만, 덕 타이핑이 추상 베이스 클래스 로 보완될 수 있음에 유의해야 합니다.) 대신에, `hasattr()` 검사나 *EAFP* 프로그래밍을 씁니다.

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

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

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

f-string (f-문자열) `'f'` 나 `'F'` 를 앞에 붙인 문자열 리터럴들을 흔히 "f-문자열"이라고 부르는데, 포맷 문자열 리터럴 의 줄임말입니다. **PEP 498** 을 보세요.

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

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

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

finder (파인더) 임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

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

더 자세한 내용은 **PEP 302**, **PEP 420**, **PEP 451** 에 나옵니다.

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

function (함수) 호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 인자 가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. 매개변수 와 메서드 와 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.

__future__ 프로그래머가 현재 인터프리터와 호환되지 않는 새 언어 기능들을 활성화할 수 있도록 하는 가상 모듈.

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

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

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

generator (제너레이터) 제너레이터 이터레이터 를 돌려주는 함수. 일반 함수처럼 보이는데, 일련의 값들을 만드는 *`yield`* 표현식을 포함한다는 점이 다릅니다. 이 값들은 for-루프로 사용하거나 `next()` 함수로 한 번에 하나씩 꺼낼 수 있습니다.

보통 제너레이터 함수를 가리키지만, 어떤 문맥에서는 제너레이터 이터레이터 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

generator iterator (제너레이터 이터레이터) 제너레이터 함수가 만드는 객체.

각 *`yield`*는 일시적으로 처리를 중단하고, 그 위치의 (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 제너레이터 이터레이터 가 재개되면, 떠난 곳으로 복귀합니다 (호출마다 새로 시작하는 함수와 대비됩니다).

generator expression (제너레이터 표현식) 이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 `for` 절과 생략 가능한 `if` 절이 뒤에 붙는 일반 표현식 처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

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

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

싱글 디스패치 용어집 항목과 `functools.singledispatch()` 데코레이터와 **PEP 443**도 보세요.

GIL 전역 인터프리터 록 을 보세요.

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

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

(훨씬 더 미세하게 공유 데이터를 잠그는) "스레드에 자유로운(free-threaded)" 인터프리터를 만들고자 하는 과거의 노력은 성공적이지 못했는데, 흔한 단일 프로세서 경우의 성능 저하가 심하기 때문입

니다. 이 성능 이슈를 극복하는 것은 구현을 훨씬 복잡하게 만들어서 유지 비용이 더 들어갈 것으로 여겨지고 있습니다.

hash-based pyc (해시 기반 pyc) 유효성을 판별하기 위해 해당 소스 파일의 최종 수정 시간이 아닌 해시를 사용하는 바이트 코드 캐시 파일. 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 환경 (Integrated Development Environment). IDLE은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

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

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

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

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

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

interpreted (인터프리티드) 바이트 코드 컴파일러의 존재 때문에 그 구분이 흐릿해지기는 하지만, 파이썬은 컴파일 언어가 아니라 인터프리터 언어입니다. 이것은 명시적으로 실행 파일을 만들지 않고도, 소스 파일을 직접 실행할 수 있다는 뜻입니다. 그 프로그램이 좀 더 천천히 실행되기는 하지만, 인터프리터 언어는 보통 컴파일 언어보다 짧은 개발/디버깅 주기를 갖습니다. 대화형 도 보세요.

interpreter shutdown (인터프리터 종료) 종료하라는 요청을 받을 때, 파이썬 인터프리터는 특별한 시기에 진입하는데, 모듈이나 여러 가지 중요한 내부 구조들과 같은 모든 할당된 자원들을 단계적으로 반납합니다. 또한, 가비지 수거기 를 여러 번 호출합니다. 사용자 정의 파괴자나 weakref 콜백에 있는 코드들의 실행을 시작시킬 수 있습니다. 종료 시기 동안 실행되는 코드는 다양한 예외들을 만날 수 있는데, 그것이 의존하는 자원들이 더 기능하지 않을 수 있기 때문입니다 (흔한 예는 라이브러리 모듈이나 경고 장치들입니다).

인터프리터 종료의 주된 원인은 실행되는 `__main__` 모듈이나 스크립트가 실행을 끝내는 것입니다.

iterable (이터러블) 멤버들을 한 번에 하나씩 돌려줄 수 있는 객체. 이터러블의 예로는 모든 (`list`, `str`, `tuple` 같은) 시퀀스 형들, `dict` 같은 몇몇 비 시퀀스 형들, 파일 객체들, *__iter__()* 나 시퀀스 개념을 구현하는 *__getitem__()* 메서드를 써서 정의한 모든 클래스의 객체들이 있습니다.

이터러블은 *for* 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 (`zip()`, `map()`, …) 에 사용될 수 있습니다. 이터러블 객체가 내장 함수 `iter()` 에 인자로 전달되면, 그 객체의 이터레이터를 돌려줍니다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효합니다. 이터러블을 사용할 때, 보통은 `iter()` 를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없습니다. `for` 문은 이것들을 여러분을 대신해서 자동으로 해주는데, 루프를 도는 동안 이터레이터를 잡아둘 이름 없는 변수를 만듭니다. 이터레이터, 시퀀스, 제너레이터 도 보세요.

iterator (이터레이터) 데이터의 스트림을 표현하는 객체. 이터레이터의 `__next__()` 메서드를 반복적으로 호출하면 (또는 내장 함수 `next()` 로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 `StopIteration` 예외를 일으킵니다. 이 지점에서, 이터레이터 객

체는 소진되고, 이후의 모든 `__next__()` 메서드 호출은 `StopIteration` 예외를 다시 일으키기만 합니다. 이터레이터는 이터레이터 객체 자신을 돌려주는 *`__iter__()`* 메서드를 가질 것이 요구되기 때문에, 이터레이터는 이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드입니다. (`list` 같은) 컨테이너 객체는 `iter()` 함수로 전달하거나 *`for`* 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

typeiter 에 더 자세한 내용이 있습니다.

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

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

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

keyword argument (키워드 인자) 인자 를 보세요.

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

LBYL 뛰기 전에 보라(Look before you leap). 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 사전 조건들을 검사합니다. 이 스타일은 *EAFP* 접근법과 대비되고, 많은 *`if`* 문의 존재로 특징지어집니다.

다중 스레드 환경에서, LBYL 접근법은 "보기"와 "뛰기" 간에 경쟁 조건을 만들게 될 위험이 있습니다. 예를 들어, 코드 `if key in mapping: return mapping[key]` 는 검사 후에, 하지만 조회 전에, 다른 스레드가 *key*를 *mapping*에서 제거하면 실패할 수 있습니다. 이런 이슈는 록이나 EAFP 접근법을 사용함으로써 해결될 수 있습니다.

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

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

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

magic method (매직 메서드) 특수 메서드 의 비공식적인 비슷한 말.

mapping (매핑) 임의의 키 조회를 지원하고 `Mapping` 이나 `MutableMapping` 추상 베이스 클래스에 지정된 메서드들을 구현하는 컨테이너 객체. 예로는 `dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` 를 들 수 있습니다.

meta path finder (메타 경로 파인더) `sys.meta_path` 의 검색이 돌려주는 파인더. 메타 경로 파인더는 경로 엔트리 파인더 와 관련되어 있기는 하지만 다릅니다.

메타 경로 파인더가 구현하는 메서드들에 대해서는 `importlib.abc.MetaPathFinder` 를 보면 됩니다.

metaclass (메타 클래스) 클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 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 The Python 2.3 Method Resolution Order를 보면 됩니다.

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

패키지 도 보세요.

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

MRO 메서드 결정 순서 를 보세요.

mutable (가변) 가변 객체는 값이 변할 수 있지만 `id()` 는 일정하게 유지합니다. 불변 도 보세요.

named tuple (네임드 튜플) The term "named tuple" applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand or it can be created with the factory function `collections.namedtuple()`. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.

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

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

모듈 도 보세요.

nested scope (중첩된 스코프) 둘러싼 정의에서 변수를 참조하는 능력. 예를 들어, 다른 함수 내부에서 정의된 함수는 바깥 함수에 있는 변수들을 참조할 수 있습니다. 중첩된 스코프는 기본적으로는 참조만 가능할 뿐, 대입은 되지 않는다는 것에 주의해야 합니다. 지역 변수들은 가장 내부의 스코프에서 읽고 씁니다. 마찬가지로, 전역 변수들은 전역 이름 공간에서 읽고 씁니다. *nonlocal* 은 바깥 스코프에 쓰는 것을 허락합니다.

new-style class (뉴스타일 클래스) 지금은 모든 클래스 객체에 사용되고 있는 클래스 버전의 예전 이름. 초기의 파이썬 버전에서는, 오직 뉴스타일 클래스만 *__slots__*, 디스크립터, 프라퍼티,

__getattribute__(), 클래스 메서드, 스태틱 메서드와 같은 파이썬의 새롭고 다양한 기능들을 사용할 수 있었습니다.

object (객체) 상태 (어트리뷰트나 값) 를 갖고 동작 (메서드) 이 정의된 모든 데이터. 또한, 모든 뉴스타일 클래스 의 최종적인 베이스 클래스입니다.

package (패키지) 서브 모듈들이나, 재귀적으로 서브 패키지들을 포함할 수 있는 파이썬 모듈. 기술적으로, 패키지는 `__path__` 어트리뷰트가 있는 파이썬 모듈입니다.

정규 패키지 와 이름 공간 패키지 도 보세요.

parameter (매개변수) 함수 (또는 메서드) 정의에서 함수가 받을 수 있는 인자 (또는 어떤 경우 인자들) 를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있습니다:

- 위치-키워드 *(positional-or-keyword)*: 위치 인자 나 키워드 인자 로 전달될 수 있는 인자를 지정합니다. 이것이 기본 형태의 매개변수입니다, 예를 들어 다음에서 *foo* 와 *bar*:

```
def func(foo, bar=None): ...
```

- 위치-전용 *(positional-only)*: 위치로만 제공될 수 있는 인자를 지정합니다. 파이썬은 위치-전용 매개변수를 정의하는 문법을 갖고 있지 않습니다. 하지만, 어떤 매장 함수들은 위치-전용 매개변수를 갖습니다 (예를 들어, `abs()`).
- 키워드-전용 *(keyword-only)*: 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 `*`를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 *kw_only1* 와 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- 가변-위치 *(var-positional)*: (다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정합니다. 이런 매개변수는 매개변수 이름에 `*` 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 다음에서 *args*:

```
def func(*args, **kwargs): ...
```

- 가변-키워드 *(var-keyword)*: (다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정합니다. 이런 매개변수는 매개변수 이름에 `**`를 앞에 붙여서 정의될 수 있습니다, 예를 들어 위의 예에서 *kwargs*.

매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있습니다.

인자 용어집 항목, 인자와 매개변수의 차이에 나오는 FAQ 질문, `inspect.Parameter` 클래스, 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_hook` 리스트에 있는 콜러블인데, 특정 경로 엔트리 에서 모듈을 찾는 법을 알고 있다면 경로 엔트리 파인더 를 돌려줍니다.

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

path-like object (경로류 객체) 파일 시스템 경로를 나타내는 객체. 경로류 객체는 경로를 나타내는 `str` 나 `bytes` 객체이거나 `os.PathLike` 프로토콜을 구현하는 객체입니다. `os.PathLike` 프로토콜을 지원하는 객체는 `os.fspath()` 함수를 호출해서 `str` 나 `bytes` 파일 시스템 경로로 변환될 수 있습니다; 대신 `os.fsdecode()` 와 `os.fsencode()` 는 각각 `str` 나 `bytes` 결과를 보장하는데 사용될 수 있습니다. **PEP 519**로 도입되었습니다.

PEP 파이썬 개선 제안. PEP는 파이썬 커뮤니티에 정보를 제공하거나 파이썬 또는 그 프로세스 또는 환경에 대한 새로운 기능을 설명하는 설계 문서입니다. PEP는 제안된 기능에 대한 간결한 기술 사양 및 근거를 제공해야 합니다.

PEP는 주요 새로운 기능을 제안하고 문제에 대한 커뮤니티 입력을 수집하며 파이썬에 들어간 설계 결정을 문서로 만들기 위한 기본 메커니즘입니다. PEP 작성자는 커뮤니티 내에서 합의를 구축하고 반대 의견을 문서화 할 책임이 있습니다.

PEP 1 참조하세요.

portion (포션) PEP 420 에서 정의한 것처럼, 이름 공간 패키지에 이바지하는 하나의 디렉터리에 들어있는 파일들의 집합 (zip 파일에 저장되는 것도 가능합니다).

positional argument (위치 인자) 인자 를 보세요.

provisional API (잠정 API) 잠정 API는 표준 라이브러리의 과거 호환성 보장으로부터 신중히 제외된 것입니다. 인터페이스의 큰 변화가 예상되지는 않지만, 잠정적이라고 표시되는 한, 코어 개발자들이 필요하다고 생각한다면 과거 호환성이 유지되지 않는 변경이 일어날 수 있습니다. 그런 변경은 불필요한 방식으로 일어나지는 않을 것입니다 — API를 포함하기 전에 놓친 중대하고 근본적인 결함이 발견된 경우에만 일어날 것입니다.

잠정 API에서조차도, 과거 호환성이 유지되지 않는 변경은 "최후의 수단"으로 여겨집니다 - 모든 식별된 문제들에 대해 과거 호환성을 유지하는 해법을 찾으려는 모든 시도가 선행됩니다.

이 절차는 표준 라이브러리가 오랜 시간 동안 잘못된 설계 오류에 발목 잡히지 않고 발전할 수 있도록 만듭니다. 더 자세한 내용은 PEP 411을 보면 됩니다.

provisional package (잠정 패키지) 잠정 *API* 를 보세요.

Python 3000 (파이썬 3000) 파이썬 3.x 배포 라인의 별명 (버전 3의 배포가 먼 미래의 이야기던 시절에 만들어진 이름이다.) 이것을 "Py3k" 로 줄여 쓰기도 합니다.

Pythonic (파이썬다운) 다른 언어들에서 일반적인 개념들을 사용해서 코드를 구현하는 대신, 파이썬 언어에서 가장 자주 사용되는 이디엄들을 가까이 따르는 아이디어나 코드 조각. 예를 들어, 파이썬에서 자주 쓰는 이디엄은 *for* 문을 사용해서 이터러블의 모든 요소로 루핑하는 것입니다. 다른 많은 언어에는 이런 종류의 구성물이 없으므로, 파이썬에 익숙하지 않은 사람들은 대신에 숫자 카운터를 사용하기도 합니다:

```
for i in range(len(food)):
    print(food[i])
```

더 깔끔한, 파이썬다운 방법은 이렇습니다:

```
for piece in food:
    print(piece)
```

qualified name (정규화된 이름) 모듈의 전역 스코프에서 모듈에 정의된 클래스, 함수, 메서드에 이르는 "경로"를 보여주는 점으로 구분된 이름. PEP 3155 에서 정의됩니다. 최상위 함수와 클래스의 경우에, 정규화된 이름은 객체의 이름과 같습니다:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

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

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

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

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

이름 공간 패키지 도 보세요.

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

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

collections.abc.Sequence 추상 베이스 클래스는 *__getitem__()* 과 *__len__()*을 넘어서 훨씬 풍부한 인터페이스를 정의하는데, count(), index(), *__contains__()*, *__reversed__()*를 추가합니다. 이 확장된 인터페이스를 구현한 형을 register()를 사용해서 명시적으로 등록할 수 있습니다.

single dispatch (싱글 디스패치) 구현이 하나의 인자의 형에 기초해서 결정되는 제네릭 함수 디스패치의 한 형태.

slice (슬라이스) 보통 시퀀스 의 일부를 포함하는 객체. 슬라이스는 서브 스크립트 표기법을 사용해서 만듭니다. variable_name[1:3:5] 처럼, [] 안에서 여러 개의 숫자를 콜론으로 분리합니다. 대괄호 (서브 스크립트) 표기법은 내부적으로 slice 객체를 사용합니다.

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

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

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

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

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

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

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

형 에일리어스는 형 힌트를 단순화하는 데 유용합니다. 예를 들면:

```
from typing import List, Tuple

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

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

이 기능을 설명하는 typing과 **PEP 484**를 참조하세요.

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

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

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

이 기능을 설명하는 typing과 **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.

virtual environment (가상 환경) 파이썬 사용자와 응용 프로그램이, 같은 시스템에서 실행되는 다른 파이썬 응용 프로그램들의 동작에 영향을 주지 않으면서, 파이썬 배포 패키지들을 설치하거나 업그레이드하는 것을 가능하게 하는, 협력적으로 격리된 실행 환경.

venv 도 보세요.

virtual machine (가상 기계) 소프트웨어만으로 정의된 컴퓨터. 파이썬의 가상 기계는 바이트 코드 컴파일러가 출력하는 바이트 코드를 실행합니다.

Zen of Python (파이썬 젠) 파이썬 디자인 원리와 철학들의 목록인데, 인어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 "import this" 를 입력하면 보입니다.

APPENDIX B

이 설명서에 관하여

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

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

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 작가;
- reStructuredText와 Docutils 스위트를 만드는 Docutils 프로젝트.
- Fredrik Lundh, 그의 Alternative Python Reference 프로젝트에서 Sphinx가 많은 아이디어를 얻었습니다.

B.1 파이썬 설명서의 공헌자들

많은 사람이 파이썬 언어, 파이썬 표준 라이브러리 및 파이썬 설명서에 기여했습니다. 기여자의 부분적인 목록은 파이썬 소스 배포판의 Misc/ACKS 를 참조하십시오.

파이썬이 이런 멋진 설명서를 갖게 된 것은 파이썬 커뮤니티의 입력과 기여 때문입니다 – 감사합니다!

APPENDIX C

역사와 라이센스

C.1 소프트웨어의 역사

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

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

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

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

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

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

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

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.16

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.7.16 software in source or binary form and its associated␣
↪documentation.

2. Subject to the terms and conditions of this License Agreement, PSF␣
↪hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative␣
↪works,
   distribute, and otherwise use Python 3.7.16 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's␣
↪notice of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.7.16 alone or in any derivative␣
↪version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.7.16 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made␣
↪to Python
   3.7.16.

4. PSF is making Python 3.7.16 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY␣
↪OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY␣
↪REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR␣
↪THAT THE
   USE OF PYTHON 3.7.16 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.16
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.16, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

```
6. This License Agreement will automatically terminate upon a material␣
↪breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee. ␣
↪This License
   Agreement does not grant permission to use PSF trademarks or trade name␣
↪in a
   trademark sense to endorse or promote products or services of Licensee,␣
↪or any
   third party.

8. By copying, installing or otherwise using Python 3.7.16, 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

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

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

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

C.3 포함된 소프트웨어에 대한 라이센스 및 승인

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

C.3.1 메르센 트위스터

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

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

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

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

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

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

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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


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

C.3.2 소켓

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

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

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

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

C.3.3 비동기 소켓 서비스

asynchat과 asyncore 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

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

C.3.4 쿠키 관리

`http.cookies` 모듈은 다음과 같은 주의 사항을 포함합니다:

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

               All Rights Reserved

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

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

C.3.5 실행 추적

`trace` 모듈은 다음과 같은 주의 사항을 포함합니다:

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

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

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

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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


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

C.3.6 UUencode 및 UUdecode 함수

uu 모듈은 다음과 같은 주의 사항을 포함합니다:

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

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

C.3.7 XML 원격 프로시저 호출

xmlrpc.client 모듈은 다음과 같은 주의 사항을 포함합니다:

```
    The XML-RPC client interface is

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

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

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

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

C.3.8 test_epoll

`test_epoll` 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

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

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

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

C.3.9 Select kqueue

`select` 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

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

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.10 SipHash24

파일 `Python/pyhash.c` 에는 Dan Bernstein의 SipHash24 알고리즘의 Marek Majkowski의 구현이 포함되어 있습니다. 여기에는 다음과 같은 내용이 포함되어 있습니다:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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

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

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 와 dtoa

C double과 문자열 간의 변환을 위한 C 함수 dtoa 와 strtod 를 제공하는 파일 `Python/dtoa.c` 는 현재 http://www.netlib.org/fp/ 에서 얻을 수 있는 David M. Gay의 같은 이름의 파일에서 파생되었습니다. 2009년 3월 16일에 받은 원본 파일에는 다음과 같은 저작권 및 라이센스 공지가 포함되어 있습니다:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
* THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
* WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
***********************************************************/
```

C.3.12 OpenSSL

모듈 hashlib, posix, ssl, crypt 는 운영 체제가 사용할 수 있게 하면 추가의 성능을 위해 OpenSSL 라이브러리를 사용합니다. 또한, 윈도우와 맥 OS X 파이썬 설치 프로그램은 OpenSSL 라이브러리 사본을 포함할 수 있으므로, 여기에 OpenSSL 라이센스 사본을 포함합니다:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

pyexpat 확장은 빌드를 --with-system-expat 로 구성하지 않는 한, 포함된 expat 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

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

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

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

C.3.14 libffi

_ctypes 확장은 빌드를 --with-system-libffi 로 구성하지 않는 한, 포함된 libffi 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

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

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

C.3.15 zlib

`zlib` 확장은 시스템에서 발견된 zlib 버전이 너무 오래되어서 빌드에 사용될 수 없으면, 포함된 zlib 소스 사본을 사용하여 빌드됩니다:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

`tracemalloc` 에 의해 사용되는 해시 테이블의 구현은 cfuhash 프로젝트를 기반으로 합니다:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

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

C.3.17 libmpdec

_decimal 모듈은 빌드를 --with-system-libmpdec 로 구성하지 않는 한, 포함된 libmpdec 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

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

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

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

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

APPENDIX D

저작권

파이썬과 이 설명서는:

전체 라이센스 및 사용 권한 정보는 역사와 라이센스 에서 제공합니다.

색인

Non-alphabetical

A

B

C

D

E

F

G

H

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z