The Python Language Reference

출시 버전 *2.7.18*

Guido van Rossum
and the Python development team

5월 20, 2020

Contents

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

CHAPTER 1

개요

이 레퍼런스 설명서는 파이썬 프로그래밍 언어를 설명한다. 자습서를 목표로 하고 있지 않다.

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

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation in widespread use (although alternate implementations exist), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short《implementation notes》sprinkled throughout the text.

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

1.1 대안 구현들

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

알려진 구현들은:

CPython 원조이기도 하고 가장 잘 관리되고 있는 C로 작성된 파이썬 구현이다. 언어의 새로운 기능은 보통 여기에서 처음 등장한다.

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

Python for .NET 이 구현은 실제로는 CPython 구현을 사용하지만, 매니지드(managed) .NET 응용 프로그램이고 .NET 라이브러리를 제공한다. Bryan Lloyd가 만들었다. 더 자세한 정보는 Python for .NET 홈페이지 에서 제공된다.

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

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

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

1.2 표기법

구문 분석과 문법의 기술은 수정된 BNF 문법 표기법을 사용한다. 이것은 다음과 같은 정의 스타일을 사용한다.

```
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 말한다. 한편 `lc_letter` 는 `'a'` 와 `'z'` 사이의 문자 하나다. (사실 이 규칙은 이 문서에서 구문과 문법 규칙에서 정의되는 이름들에 대한 규칙이다.)

개별 규칙은 이름 (위 규칙에 등장하는 `name`) 과 `::=` 로 시작한다. 세로막대(`|`)는 대안들을 분리하는 데 사용된다; 이 표기법에서 우선순위가 가장 낮은 연산자다. 별표(`*`)는 앞에 나오는 항목이 생략되거나 한 번 이상 반복될 수 있다는 의미다; 비슷하게, 더하기(`+`)는 한 번 이상 반복될 수 있지만 생략할 수는 없다는 뜻이고, 꺾쇠괄호(`[]`)로 둘러싸인 것은 최대 한 번 나올 수 있고, 생략 가능하다는 뜻이다. `*` 와 `+` 연산자는 최대한 엄격하게 연결된다; 우선순위가 가장 높다; 괄호는 덩어리로 묶는 데 사용된다. 문자열 리터럴은 따옴표로 둘러싸인다. 공백은 토큰을 분리하는 용도로만 사용된다. 규칙은 보통 한 줄로 표현된다; 대안이 많은 규칙은 여러 줄로 표현될 수도 있는데, 뒤따르는 줄들이 세로막대로 시작되게 만든다.

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

사용되는 표기법이 거의 같다고 하더라도, 구문과 문법 정의 간에는 커다란 차이가 있다: 구문 정의는 입력의 개별 문자에 적용되는 반면, 문법 정의는 구문 분석기가 만들어내는 토큰들에 적용된다. 다음 장 (《구문 분석 (Lexical Analysis)》) 에서 사용되는 모든 BNF는 구문 정의다; 그 이후의 장에서는 문법 정의다.

CHAPTER 2

구문 분석

파이썬 프로그램은 파서*(parser)* 에 의해 읽힌다. 파서의 입력은 구문 분석기*(lexical analyzer)* 가 만들어내는 토큰*(token)* 들의 스트림이다. 이 장에서는 구문 분석기가 어떻게 파일을 토큰들로 분해하는지 설명한다.

Python uses the 7-bit ASCII character set for program text.

버전 2.3에 추가: An encoding declaration can be used to indicate that string literals and comments use an encoding different from ASCII.

For compatibility with older versions, Python only warns if it finds 8-bit characters; those warnings should be corrected by either declaring an explicit encoding, or using escape sequences if those bytes are binary data, instead of characters.

The run-time character set depends on the I/O devices connected to the program but is generally a superset of ASCII.

Future compatibility note: It may be tempting to assume that the character set for 8-bit characters is ISO Latin-1 (an ASCII superset that covers most western languages that use the Latin alphabet), but it is possible that in the future Unicode text editors will become common. These generally use the UTF-8 encoding, which is also an ASCII superset, but with very different use for the characters with ordinals 128-255. While there is no consensus on this subject yet, it is unwise to assume either Latin-1 or UTF-8, even though the current implementation appears to favor Latin-1. This applies both to the source character set and the run-time character set.

2.1 줄 구조(Line structure)

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

2.1.1 논리적인 줄

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

2.1.2 물리적인 줄

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files and strings, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform. The end of input also serves as an implicit terminator for the final physical line.

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

2.1.3 주석

주석은 문자열 리터럴에 포함되지 않는 해시 문자(#)로 시작하고 물리적인 줄의 끝에서 끝난다. 묵시적인 줄 결합 규칙이 유효하지 않은 이상, 주석은 논리적인 줄을 종료시킨다. 주석은 문법이 무시한다; 토큰으로 만들어지지 않는다.

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 처리된다. 이 정규식의 첫 번째 그룹은 소스 코드 파일의 인코딩 이름을 지정한다. 인코딩 선언은 줄 전체에 홀로 나와야 한다. 만약 두 번째 줄이라면, 첫 번째 줄 역시 주석만 있어야 한다. 인코딩 선언의 권장 형태는 두 개다. 하나는

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

인데 GNU Emacs에서도 인식된다. 다른 하나는

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

which is recognized by Bram Moolenaar's VIM. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`'\xef\xbb\xbf'`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft's **notepad**).

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, in particular to find the end of a string, and to interpret the contents of Unicode literals. String literals are converted to Unicode for syntactical analysis, then converted back to their original encoding before interpretation starts.

2.1.5 명시적인 줄 결합

둘 이상의 물리적인 줄은 역 슬래시 문자(\)를 사용해서 논리적인 줄로 결합할 수 있다: 물리적인 줄이 문자열 리터럴이나 주석의 일부가 아닌 역 슬래시 문자로 끝나면, 역 슬래시와 뒤따르는 개행 문자가 제거된 채로, 현재 만들어지고 있는 논리적인 줄에 합쳐진다. 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 없다. 역 슬래시는 주석을 결합하지 못한다. 역 슬래시는 문자열 리터럴을 제외한 어떤 토큰도 결합하지 못한다(즉, 문자열 리터럴 이외의 어떤 토큰도 역 슬래시를 사용해서 두 줄에 나누어 기록할 수 없다.). 문자열 리터럴 밖에 있는 역 슬래시가 앞에서 언급한 장소 이외의 곳에 등장하는 것은 문법에 어긋난다.

2.1.6 묵시적인 줄 결합

괄호(()), 꺾쇠괄호([]), 중괄호({})가 사용되는 표현은 역 슬래시 없이도 여러 개의 물리적인 줄로 나눌 수 있다. 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 있다. 이어지는 줄들의 들여쓰기는 중요하지 않다. 중간에 빈 줄이 들어가도 된다. 묵시적으로 줄 결합하는 줄 들 간에는 NEWLINE 토큰이 만들어지지 않는다. 묵시적으로 이어지는 줄들은 삼중 따옴표 된 문자열들에서도 등장할 수 있는데 (아래를 보라), 이 경우는 주석이 포함될 수 없다.

2.1.7 빈 줄

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

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 된다.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

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

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

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

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

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

```
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 보여준다:

```
 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 감지한다. 단지 마지막 에러만 구문 분석기가 감지한다. — `return r`의 들여쓰기가 스택에 있는 값과 일치하지 않는다.)

2.1.9 토큰 사이의 공백

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

2.2 다른 토큰들

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

2.3 식별자와 키워드

Identifiers (also referred to as *names*) are described by the following lexical definitions:

```
identifier  ::=  (letter|"_") (letter | digit | "_")*
letter      ::=  lowercase | uppercase
lowercase   ::=  "a"..."z"
uppercase   ::=  "A"..."Z"
digit       ::=  "0"..."9"
```

식별자는 길이에 제한이 없고, 케이스(case)는 구분된다.

2.3.1 키워드

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

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

버전 2.4에서 변경: `None` became a constant and is now recognized by the compiler as a name for the built-in object `None`. Although it is not a keyword, you cannot assign a different object to it.

버전 2.5에서 변경: Using *as* and *with* as identifiers triggers a warning. To use them as keywords, enable the `with_statement` future feature .

버전 2.6에서 변경: *as* and *with* are full keywords.

2.3.2 식별자의 예약 영역

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

_* Not imported by `from module import *`. The special identifier _ is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `__builtin__` module. When not in interactive mode, _ has no special meaning and is not defined. See section 임포트 *(import)* 문.

참고: 이름 _ 은 종종 국제화(internationalization)와 관련되어 사용된다. 이 관례에 관해서는 `gettext` 모듈의 문서를 참조하라.

__*__ 시스템 정의 이름. 이 이름들은 인터프리터와 그 구현(표준 라이브러리를 포함한다)이 정의한다. 현재 정의된 시스템 이름은 특수 메서드 이름들 섹션과 그 외의 곳에서 논의된다. 파이썬의 미래 버전에서는 더 많은 것들이 정의될 가능성이 크다. 어떤 문맥에서건, 명시적으로 문서로 만들어진 사용법을 벗어나는 __*__ 이름의 모든 사용은, 경고 없이 손상될 수 있다.

__* 클래스-비공개 이름. 이 부류의 이름들을 클래스 정의 문맥에서 사용하면 뒤섞인 형태로 변형된다. 부모 클래스와 자식 클래스의 《비공개(private)》 어트리뷰트 간의 이름 충돌을 피하기 위함이다. 식별자 *(이름)* 섹션을 보라.

2.4 리터럴

리터럴(literal)은 몇몇 내장형들의 상숫값을 위한 표기법이다.

2.4.1 String literals

문자열 리터럴은 다음과 같은 구문 정의로 기술된다:

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"
                     | "b" | "B" | "br" | "Br" | "bR" | "BR"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''"
                     | '"""' longstringitem* '"""'
shortstringitem ::=  shortstringchar | escapeseq
longstringitem  ::=  longstringchar | escapeseq
shortstringchar ::=  <any source character except "\" or newline or the quote>
longstringchar  ::=  <any source character except "\">
escapeseq       ::=  "\" <any ASCII character>
```

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

In plain English: String literals can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter 'r' or 'R'; such strings are called *raw strings* and use different rules for interpreting backslash escape sequences. A prefix of 'u' or 'U' makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646.

Some additional escape sequences, described below, are available in Unicode strings. A prefix of `'b'` or `'B'` is ignored in Python 2; it indicates that the literal should become a bytes literal in Python 3 (e.g. when code is automatically converted with 2to3). A `'u'` or `'b'` prefix may be followed by an `'r'` prefix.

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A《quote》is the character used to open the string, i.e. either `'` or `"`.)

Unless an `'r'` or `'R'` prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

이스케이프 시퀀스	의미	유의 사항
`\newline`	Ignored	
`\\`	역 슬래시 (\)	
`\'`	작은따옴표 (')	
`\"`	큰따옴표 (")	
`\a`	ASCII 벨 (BEL)	
`\b`	ASCII 백스페이스 (BS)	
`\f`	ASCII 폼 피드 (FF)	
`\n`	ASCII 라인 피드 (LF)	
`\N{name}`	Character named *name* in the Unicode database (Unicode only)	
`\r`	ASCII 캐리지 리턴 (CR)	
`\t`	ASCII 가로 탭 (TAB)	
`\uxxxx`	Character with 16-bit hex value *xxxx* (Unicode only)	(1)
`\Uxxxxxxxx`	Character with 32-bit hex value *xxxxxxxx* (Unicode only)	(2)
`\v`	ASCII 세로 탭 (VT)	
`\ooo`	8진수 *ooo* 로 지정된 문자	(3,5)
`\xhh`	16진수 *hh* 로 지정된 문자	(4,5)

유의 사항:

(1) Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(2) Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if Python is compiled to use 16-bit code units (the default).

(3) 표준 C와 마찬가지로, 최대 세 개의 8진수가 허용된다.

(4) 표준 C와는 달리, 정확히 두 개의 16진수가 제공되어야 한다.

(5) In a string literal, hexadecimal and octal escapes denote the byte with the given value; it is not necessary that the byte encodes a character in the source character set. In a Unicode literal, these escapes denote a Unicode character with the given value.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the string*. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences marked as《(Unicode only)》in the table above fall into the category of unrecognized escapes for non-Unicode string literals.

When an `'r'` or `'R'` prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string*. For example, the string literal `r"\n"` consists of two characters: a backslash and a lowercase `'n'`. String quotes can be escaped with a backslash, but the backslash remains in the string; for example, `r"\""` is a valid string literal consisting of two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw string cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, *not* as a line continuation.

When an `'r'` or `'R'` prefix is used in conjunction with a `'u'` or `'U'` prefix, then the `\uXXXX` and `\UXXXXXXXX` escape sequences are processed while *all other backslashes are left in the string*. For example, the string literal `ur"\u0062\n"`

consists of three Unicode characters: 〈LATIN SMALL LETTER B〉, 〈REVERSE SOLIDUS〉, and 〈LATIN SMALL LETTER N〉. Backslashes can be escaped with a preceding backslash; however, both remain in the string. As a result, `\uXXXX` escape sequences are only recognized when there are an odd number of backslashes.

2.4.2 문자열 리터럴 이어붙이기

Multiple adjacent string literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, `"hello" 'world'` is equivalent to `"helloworld"`. This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example:

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

Note that this feature is defined at the syntactical level, but implemented at compile time. The 〈+〉 operator must be used to concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings).

2.4.3 숫자 리터럴

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

숫자 리터럴이 부호를 포함하지 않는 것에 주의해야 한다; `-1` 과 같은 구문은 일 항 연산자 〈-〈 과 리터럴 `1` 로 구성된 표현식이다.

2.4.4 Integer and long integer literals

Integer and long integer literals are described by the following lexical definitions:

```
longinteger     ::=   integer ("l" | "L")
integer         ::=   decimalinteger | octinteger | hexinteger | bininteger
decimalinteger  ::=   nonzerodigit digit* | "0"
octinteger      ::=   "0" ("o" | "O") octdigit+ | "0" octdigit+
hexinteger      ::=   "0" ("x" | "X") hexdigit+
bininteger      ::=   "0" ("b" | "B") bindigit+
nonzerodigit    ::=   "1"..."9"
octdigit        ::=   "0"..."7"
bindigit        ::=   "0" | "1"
hexdigit        ::=   digit | "a"..."f" | "A"..."F"
```

Although both lower case `'l'` and upper case `'L'` are allowed as suffix for long integers, it is strongly recommended to always use `'L'`, since the letter `'l'` looks too much like the digit `'1'`.

Plain integer literals that are above the largest representable plain integer (e.g., 2147483647 when using 32-bit arithmetic) are accepted as if they were long integers instead.[1] There is no limit for long integer literals apart from what can be stored in available memory.

[1] In versions of Python prior to 2.4, octal and hexadecimal literals in the range just above the largest representable plain integer but below the largest unsigned 32-bit number (on a machine using 32-bit arithmetic), 4294967296, were taken as the negative plain integer obtained by subtracting 4294967296 from their unsigned value.

Some examples of plain integer literals (first row) and long integer literals (second and third rows):

```
7     2147483647                        0177
3L    79228162514264337593543950336L    0377L   0x100000000L
      79228162514264337593543950336             0xdeadbeef
```

2.4.5 실수 리터럴

실수 리터럴은 다음과 같은 구문 정의로 표현된다:

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

Note that the integer and exponent parts of floating point numbers can look like octal integers, but are interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals:

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

Note that numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator `-` and the literal `1`.

2.4.6 허수 리터럴

허수 리터럴은 다음과 같은 구문 정의로 표현된다:

```
imagnumber  ::=   (floatnumber | intpart) ("j" | "J")
```

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

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

2.5 연산자

다음과 같은 토큰들은 연산자다:

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

The comparison operators `<>` and `!=` are alternate spellings of the same operator. `!=` is the preferred spelling; `<>` is obsolescent.

2.6 구분자

다음 토큰들은 문법에서 구분자(delimiter)로 기능한다:

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

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis in slices. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

다음의 인쇄되는 ASCII 문자들은 다른 토큰들 일부로서 특별한 의미를 갖거나, 그 밖의 경우 구문 분석기에 유의미하다:

```
'       "       #       \
```

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

```
$       ?
```

CHAPTER 3

데이터 모델

3.1 객체, 값, 형

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

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The 〈`is`〉 operator compares the identity of two objects; the `id()` function returns an integer representing its identity (currently implemented as its address). An object's *type* is also unchangeable.[1] An object's type determines the operations that the object supports (e.g., 《does it have a length?》) and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter's value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

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

CPython implementation detail: CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (ex: always close files).

구현이 제공하는 추적이나 디버깅 장치의 사용은 그렇지 않으면 수거될 수 있는 객체들을 살아있도록 만들 수 있음에 주의해야 한다. 또한 〈`try`…`except`〉 문으로 예외를 잡는 것도 객체를 살아있게 만들 수 있다.

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

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

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

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

3.2 표준형 계층

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.).

아래에 나오는 몇몇 형에 대한 설명은 〈특수 어트리뷰트(special attribute)〉 를 나열하는 문단을 포함한다. 이것들은 구현에 접근할 방법을 제공하는데, 일반적인 사용을 위한 것이 아니다. 정의는 앞으로 변경될 수 있다.

None 이 형은 하나의 값만을 갖는다. 이 값을 갖는 하나의 객체가 존재한다. 이 객체에는 내장된 이름 `None` 을 통해 접근한다. 여러 가지 상황에서 값의 부재를 알리는 데 사용된다. 예를 들어, 명시적으로 뭔가를 돌려주지 않는 함수의 반환 값이다. 논리값은 거짓이다.

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

Ellipsis This type has a single value. There is a single object with this value. This object is accessed through the built-in name `Ellipsis`. It is used to indicate the presence of the `...` syntax in a slice. Its truth value is true.

`numbers.Number` 이것들은 숫자 리터럴에 의해 만들어지고, 산순 연산과 내장 산술 함수들이 결과로 돌려준다. 숫자 객체는 불변이다; 한 번 값이 만들어지면 절대 변하지 않는다. 파이썬의 숫자는 당연히 수학적인 숫자들과 밀접하게 관련되어 있다, 하지만 컴퓨터의 숫자 표현상의 제약을 받고 있다.

파이썬은 정수, 실수, 복소수를 구분한다:

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

There are three types of integers:

Plain integers These represent numbers in the range -2147483648 through 2147483647. (The range may be larger on machines with a larger natural word size, but not smaller.) When the result of an operation would fall outside this range, the result is normally returned as a long integer (in some cases, the exception `OverflowError` is raised instead). For the purpose of shift and mask operations, integers are assumed to have a binary, 2's complement notation using 32 or more bits, and hiding no bits from the user (i.e., all 4294967296 different bit patterns correspond to different values).

Long integers 이것은 (가상) 메모리가 허락하는 한, 제약 없는 범위의 숫자를 표현한다. 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 준다.

Booleans These represent the truth values False and True. The two objects representing the values `False` and `True` are the only Boolean objects. The Boolean type is a subtype of plain integers, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings `"False"` or `"True"` are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. Any operation, if it yields a result in the plain integer domain, will yield the same result in the long integer domain or when using mixed operands. The switch between domains is transparent to the programmer.

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

numbers.Complex 이것들은 기계 수준 배정도 부동 소수점 수의 쌍으로 복소수를 나타낸다. 부동 소수점 수와 한계와 문제점을 공유한다. 복소수 `z` 의 실수부와 허수부는, 읽기 전용 어트리뷰트 `z.real` 와 `z.imag` 로 꺼낼 수 있다.

시퀀스들 음이 아닌 정수로 인덱싱 (indexing) 될 수 있는 유한한 길이의 순서 있는 집합을 나타낸다. 내장함수 `len()` 은 시퀀스가 가진 항목들의 개수를 돌려준다. 시퀀스의 길이가 *n* 일 때, 인덱스 (index) 집합은 숫자 0, 1, …, *n*-1을 포함한다. 시퀀스 *a* 의 항목 *i* 는 `a[i]` 로 선택된다.

시퀀스는 슬라이싱도 지원한다: `a[i:j]` 는 $i <= k < j$ 를 만족하는 모든 항목 *k* 를 선택한다. 표현식에서 사용될 때, 슬라이스는 같은 형의 시퀀스다. 인덱스 집합은 0에서 시작되도록 다시 번호 매겨진다.

어떤 시퀀스는 세 번째 《스텝 (step)》 파라미터를 사용하는 《확장 슬라이싱 (extended slicing)》 도 지원한다: `a[i:j:k]` 는 `x = i + n*k`, $n >= 0$, $i <= x < j$ 를 만족하는 모든 항목 *x* 를 선택한다.

시퀀스는 불변성에 따라 구분된다

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

다음과 같은 형들은 불변 시퀀스다:

문자열 (Strings) The items of a string are characters. There is no separate character type; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0–127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program. The string data type is also used to represent arrays of bytes, e.g., to hold data read from a file.

(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule?)

Unicode The items of a Unicode object are Unicode code units. A Unicode code unit is represented by a Unicode object of one item and can hold either a 16-bit or 32-bit value representing a Unicode ordinal (the maximum value for the ordinal is given in `sys.maxunicode`, and depends on how Python is configured at compile time). Surrogate pairs may be present in the Unicode object, and will be reported as two separate items. The built-in functions `unichr()` and `ord()` convert between code units and

nonnegative integers representing the Unicode ordinals as defined in the Unicode Standard 3.0. Conversion from and to other encodings are possible through the Unicode method `encode()` and the built-in function `unicode()`.

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

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

현재 두 개의 내장 가변 시퀀스형이 있다:

리스트(Lists) 리스트의 항목은 임의의 파이썬 객체다. 리스트는 콤마로 분리된 표현식을 꺾쇠괄호 안에 넣어서 만들 수 있다. (길이 0이나 1의 리스트를 만드는데 별도의 규칙이 필요 없다.)

바이트 배열(Byte Arrays) A bytearray object is a mutable array. They are created by the built-in `bytearray()` constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable bytes objects.

The extension module `array` provides an additional example of a mutable sequence type.

집합 형들(Set types) 이것들은 중복 없는 불변 객체들의 순서 없고 유한한 집합을 나타낸다. 인덱싱할 수 없다. 하지만 이터레이트할 수 있고, 내장 함수 `len()` 은 집합 안에 있는 항목들의 개수를 돌려준다. 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 것이다.

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

현재 두 개의 내장 집합 형이 있다:

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

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

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

현재 한 개의 내장 매핑 형이 있다:

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

딕셔너리는 가변이다; `{...}` 표기법으로 만들 수 있다(딕셔너리 디스플레이 섹션을 참고하라).

The extension modules `dbm`, `gdbm`, and `bsddb` provide additional examples of mapping types.

콜러블(Callable types) 이것들은 함수 호출 연산(호출 섹션 참고)이 적용될 수 있는 형들이다:

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

특수 어트리뷰트들(Special attributes):

어트리뷰트	의미	
`__doc__` `func_doc`	The function's documentation string, or `None` if unavailable.	쓰기 가능
`__name__` `func_name`	함수의 이름	쓰기 가능
`__module__`	함수가 정의된 모듈의 이름 또는 (없는 경우) `None`	쓰기 가능
`__defaults__` `func_defaults`	A tuple containing default argument values for those arguments that have defaults, or `None` if no arguments have a default value.	쓰기 가능
`__code__` `func_code`	컴파일된 함수의 바디(body)를 나타내는 코드 객체	쓰기 가능
`__globals__` `func_globals`	함수의 전역 변수들을 가진 딕셔너리에 대한 참조 — 함수가 정의된 모듈의 전역 이름 공간(namespace)	읽기 전용
`__dict__` `func_dict`	임의의 함수 어트리뷰트를 지원하는 이름 공간.	쓰기 가능
`__closure__` `func_closure`	`None` 또는 함수의 자유 변수(free variable)들에 대한 연결을 가진 셀(cell)들의 튜플.	읽기 전용

《쓰기 가능》하다고 표시된 대부분의 어트리뷰트들은 값이 대입될 때 형을 검사한다.

버전 2.4에서 변경: `func_name` is now writable.

버전 2.6에서 변경: The double-underscore attributes `__closure__`, `__code__`, `__defaults__`, and `__globals__` were introduced as aliases for the corresponding `func_*` attributes for forwards compatibility with Python 3.

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

함수 정의에 관한 추가적인 정보를 코드 객체로부터 얻을 수 있다. 아래에 나오는 내부 형의 기술을 참고하라.

User-defined methods A user-defined method object combines a class, a class instance (or `None`) and any callable object (normally a user-defined function).

Special read-only attributes: `im_self` is the class instance object, `im_func` is the function object; `im_class` is the class of `im_self` for bound methods or the class that asked for the method for unbound methods; `__doc__` is the method's documentation (same as `im_func.__doc__`); `__name__` is the method name (same as `im_func.__name__`); `__module__` is the name of the module the method was defined in, or `None` if unavailable.

버전 2.2에서 변경: `im_self` used to refer to the class that defined the method.

버전 2.6에서 변경: For Python 3 forward-compatibility, `im_func` is also available as `__func__`, and `im_self` as `__self__`.

메서드는 기반 함수의 모든 함수 어트리뷰트들을 읽을 수 있도록 지원한다(하지만 쓰기는 지원하지 않는다).

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object, an unbound user-defined method object, or a class method object. When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object; otherwise, the original method object is used as it is.

When a user-defined method object is created by retrieving a user-defined function object from a class, its `im_self` attribute is `None` and the method object is said to be unbound. When one is created by retrieving a user-defined function object from a class via one of its instances, its `im_self` attribute is the instance, and the method object is said to be bound. In either case, the new method's `im_class` attribute is the class from which the retrieval takes place, and its `im_func` attribute is the original function object.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the `im_func` attribute of the new instance is not the original method object but its `im_func` attribute.

When a user-defined method object is created by retrieving a class method object from a class or instance, its `im_self` attribute is the class itself, and its `im_func` attribute is the function object underlying the class method.

When an unbound user-defined method object is called, the underlying function (`im_func`) is called, with the restriction that the first argument must be an instance of the proper class (`im_class`) or of a derived class thereof.

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

When a user-defined method object is derived from a class method object, the《class instance》stored in `im_self` will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

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

내장 함수(Built-in functions) 내장 함수 객체는 C 함수를 둘러싸고 있다(wrapper). 내장 함수의 예로는 `len()` 과 `math.sin()` (`math` 는 표준 내장 모듈이다) 가 있다. 인자의 개수와 형은 C 함수에 의해 결정된다. 특수 읽기 전용 어트리뷰트들: `__doc__` 은 함수의 설명 문자열 또는 없는 경우 `None` 이다; `__name__` 은 함수의 이름이다; `__self__` 는 `None` 으로 설정된다 (하지만 다음 항목을 보라); `__module__` 은 함수가 정의된 모듈의 이름이거나 없는 경우 `None` 이다.

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

Class Types Class types, or《new-style classes,》are callable. These objects normally act as factories for new

instances of themselves, but variations are possible for class types that override *__new__()*. The arguments of the call are passed to *__new__()* and, in the typical case, to *__init__()* to initialize the new instance.

Classic Classes Class objects are described below. When a class object is called, a new class instance (also described below) is created and returned. This implies a call to the class's *__init__()* method if it has one. Any arguments are passed on to the *__init__()* method. If there is no *__init__()* method, the class must be called without arguments.

클래스 인스턴스(Class instances) Class instances are described below. Class instances are callable only when the class has a *__call__()* method; `x(arguments)` is a shorthand for `x.__call__(arguments)`.

모듈(Modules) Modules are imported by the *import* statement (see section 임포트 *(import)* 문). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the func_globals attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

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

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

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

Predefined (writable) attributes: `__name__` is the module's name; `__doc__` is the module's documentation string, or `None` if unavailable; `__file__` is the pathname of the file from which the module was loaded, if it was loaded from a file. The `__file__` attribute is not present for C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

클래스(Classes) Both class types (new-style classes) and class objects (old-style/classic classes) are typically created by class definitions (see section 클래스 정의). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., `C.x` is translated to `C.__dict__["x"]` (although for new-style classes in particular there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. For old-style classes, the search is depth-first, left-to-right in the order of occurrence in the base class list. New-style classes use the more complex C3 method resolution order which behaves correctly even in the presence of 〈diamond〉 inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by new-style classes can be found in the documentation accompanying the 2.3 release at https://www.python.org/download/releases/2.3/mro/.

When a class attribute reference (for class `C`, say) would yield a user-defined function object or an unbound user-defined method object whose associated class is either `C` or one of its base classes, it is transformed into an unbound user-defined method object whose `im_class` attribute is `C`. When it would yield a class method object, it is transformed into a bound user-defined method object whose `im_self` attribute is `C`. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section 디스크립터 구현하기 for another way in which attributes retrieved from a class may differ from those actually contained in its `__dict__` (note that only new-style classes support descriptors).

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

클래스 객체는 클래스 인스턴스를 돌려주도록(아래를 보라) 호출될 수 있다(위를 보라).

Special attributes: `__name__` is the class name; `__module__` is the module name in which the class was defined; `__dict__` is the dictionary containing the class's namespace; `__bases__` is a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list; `__doc__` is the class's documentation string, or `None` if undefined.

클래스 인스턴스(Class instances) A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object or an unbound user-defined method object whose associated class is the class (call it `C`) of the instance for which the attribute reference was initiated or one of its bases, it is transformed into a bound user-defined method object whose `im_class` attribute is `C` and whose `im_self` attribute is the instance. Static method and class method objects are also transformed, as if they had been retrieved from class `C`; see above under《Classes》. See section 디스크립터 구현하기 for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's `__dict__`. If no class attribute is found, and the object's class has a *`__getattr__()`* method, that is called to satisfy the lookup.

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

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

특수 어트리뷰트들: `__dict__` 는 어트리뷰트 딕셔너리다; `__class__` 는 인스턴스의 클래스다.

Files A file object represents an open file. File objects are created by the `open()` built-in function, and also by `os.popen()`, `os.fdopen()`, and the `makefile()` method of socket objects (and perhaps by other functions or methods provided by extension modules). The objects `sys.stdin`, `sys.stdout` and `sys.stderr` are initialized to file objects corresponding to the interpreter's standard input, output and error streams. See bltin-file-objects for complete documentation of file objects.

내부 형(Internal types) 인터프리터가 내부적으로 사용하는 몇몇 형들은 사용자에게 노출된다. 인터프리터의 미래 버전에서 이들의 정의는 변경될 수 있지만, 완전함을 위해 여기서 언급한다.

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

특수 읽기 전용 어트리뷰트들: `co_name` 은 함수의 이름이다; `co_argcount` 는 위치 인자들(기본값이 있는 인자들도 포함된다)의 개수다; `co_nlocals` 는 함수가 사용하는 지역 변수들(인자들을 포함한다)의 개수다; `co_varnames` 는 지역 변수들의 이름을 담고 있는 튜플이다(인자들의 이름이 먼저 나온다); `co_cellvars` 는 중첩된 함수들이 참조하는 지역 변수들의 이름을 담고 있는 튜플이다; `co_freevars` 는 자유 변수(free variables)들의 이름을 담고 있는 튜플이다; `co_code` 는 바이트 코드 명령 시퀀스를 나타내는 문자열이다; `co_consts` 는 바이트 코드가 사용하는 리터럴을 포함하는 튜플이다; `co_names` 는 바이트 코드가 사용하는 이름들을 담고 있는 튜플이다; `co_filename` 은 컴파일된 코드를 제공한 파일의 이름이다; `co_firstlineno` 는 함수의 첫 번째 줄 번호다; `co_lnotab` 은 바이트 코드에서의 위치를 줄 번호로 매핑하는 법을 문자열로 인코딩한 값이다 (자세한 내용은 인터프리터의 소스 코드를 참고하라); `co_stacksize` 는 필요한 스택의 크기다(지역 변수를 포함한다); `co_flags` 는 인터프리터의 여러 플래그(flag)들을 정수로 인코딩한 값이다.

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

퓨처 기능 선언 (`from __future__ import division`) 또한 코드 객체가 특정 기능이 활성화된 상태에서 컴파일되었는지를 나타내기 위해 `co_flags` 의 비트들을 사용한다: 함수가 퓨처 division이 활성화된 상태에서 컴파일되었으면 비트 `0x2000` 이 설정된다; 비트 `0x10` 과 `0x1000`

는 예전 버전의 파이썬에서 사용되었다.

co_flags 의 다른 비트들은 내부 사용을 위해 예약되어 있다.

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

프레임 객체 (Frame objects) 프레임 객체는 실행 프레임 (execution frame) 을 나타낸다. 트레이스백 객체에 등장할 수 있다 (아래를 보라).

Special read-only attributes: f_back is to the previous stack frame (towards the caller), or None if this is the bottom stack frame; f_code is the code object being executed in this frame; f_locals is the dictionary used to look up local variables; f_globals is used for global variables; f_builtins is used for built-in (intrinsic) names; f_restricted is a flag indicating whether the function is executing in restricted execution mode; f_lasti gives the precise instruction (this is an index into the bytecode string of the code object).

Special writable attributes: f_trace, if not None, is a function called at the start of each source code line (this is used by the debugger); f_exc_type, f_exc_value, f_exc_traceback represent the last exception raised in the parent frame provided another exception was ever raised in the current frame (in all other cases they are None); f_lineno is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to f_lineno.

트레이스백 객체 (Traceback objects) Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section *try* 문.) It is accessible as sys.exc_traceback, and also as the third item of the tuple returned by sys.exc_info(). The latter is the preferred interface, since it works correctly when the program is using multiple threads. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as sys.last_traceback.

특수 읽기 전용 어트리뷰트들: tb_next 는 스택 트레이스의 다음 단계 (예외가 발생한 프레임 방향으로) 이거나 다음 단계가 없으면 None 이다. tb_frame 은 현 단계에서의 실행 프레임이다; tb_lineno 는 예외가 발생한 줄의 번호를 준다; tb_lasti 정확한 바이트 코드 명령을 가리킨다. 만약 예외가 except 절이나 finally 절이 없는 *try* 문에서 발생하면, 줄 번호와 트레이스백의 마지막 명령 (last instruction) 은 프레임 객체의 줄 번호와 다를 수 있다.

슬라이스 객체 (Slice objects) Slice objects are used to represent slices when *extended slice syntax* is used. This is a slice using two colons, or multiple slices or ellipses separated by commas, e.g., a[i:j:step], a[i:j, k:l], or a[..., i:j]. They are also created by the built-in slice() function.

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

슬라이스 객체는 하나의 메서드를 지원한다.

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

This method takes a single integer argument *length* and computes information about the extended slice that the slice object would describe if applied to a sequence of *length* items. It returns a tuple of three integers; respectively these are the *start* and *stop* indices and the *step* or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices.

버전 2.3 에 추가.

스태틱 메서드 객체 (Static method objects) 스태틱 메서드 객체는 위에서 설명한 함수 객체를 메서드 객체로 변환하는 과정을 방지하는 방법을 제공한다. 스태틱 메서드 객체는 다른 임의의 객체, 보통 사용자 정의 메서드를 둘러싼다. 스태틱 메서드가 클래스나 클래스 인스턴스로부터 읽힐 때 객체가 실제로 돌려주는 것은 둘러싸여 있던 객체인데, 다른 어떤 변환도 적용되지 않은 상태다. 둘러싸는

객체는 그렇더라도, 스태틱 메서드 객체 자체는 콜러블이 아니다. 스태틱 메서드 객체는 내장 `staticmethod()` 생성자로 만든다.

클래스 메서드 객체(Class method objects) 스태틱 메서드 객체처럼, 클래스 메서드 객체 역시 다른 객체를 둘러싸는데, 클래스와 클래스 인스턴스로부터 그 객체를 꺼내는 방식에 변화를 준다. 그런 조회에서 클래스 메서드 객체가 동작하는 방식에 대해서는 위 《사용자 정의 메서드(User-defined methods)》에서 설명했다. 클래스 메서드 객체는 내장 `classmethod()` 생성자로 만든다.

3.3 New-style and classic classes

Classes and instances come in two flavors: old-style (or classic) and new-style.

Up to Python 2.1 the concept of `class` was unrelated to the concept of `type`, and old-style classes were the only flavor available. For an old-style class, the statement `x.__class__` provides the class of *x*, but `type(x)` is always `<type 'instance'>`. This reflects the fact that all old-style instances, independent of their class, are implemented with a single built-in type, called `instance`.

New-style classes were introduced in Python 2.2 to unify the concepts of `class` and `type`. A new-style class is simply a user-defined type, no more, no less. If *x* is an instance of a new-style class, then `type(x)` is typically the same as `x.__class__` (although this is not guaranteed – a new-style class instance is permitted to override the value returned for `x.__class__`).

The major motivation for introducing new-style classes is to provide a unified object model with a full meta-model. It also has a number of practical benefits, like the ability to subclass most built-in types, or the introduction of 《descriptors》, which enable computed properties.

For compatibility reasons, classes are still old-style by default. New-style classes are created by specifying another new-style class (i.e. a type) as a parent class, or the 《top-level type》 `object` if no other parent is needed. The behaviour of new-style classes differs from that of old-style classes in a number of important details in addition to what `type()` returns. Some of these changes are fundamental to the new object model, like the way special methods are invoked. Others are 《fixes》 that could not be implemented before for compatibility concerns, like the method resolution order in case of multiple inheritance.

While this manual aims to provide comprehensive coverage of Python's class mechanics, it may still be lacking in some areas when it comes to its coverage of new-style classes. Please see https://www.python.org/doc/newstyle/ for sources of additional information.

Old-style classes are removed in Python 3, leaving only new-style classes.

3.4 특수 메서드 이름들

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

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

3.4.1 기본적인 커스터마이제이션

`object.`**`__new__`**`(`*cls*`[, ...])`
클래스 *cls* 의 새 인스턴스를 만들기 위해 호출된다. *__new__()* 는 스태틱 메서드다 (그렇게 선언하지 않아도 되는 특별한 경우다) 인데, 첫 번째 인자로 만들려고 하는 인스턴스의 클래스가 전달된다. 나머지 인자들은 객체 생성자 표현 (클래스 호출) 에 전달된 것들이다. *__new__()* 의 반환 값은 새 객체 인스턴스이어야 한다 (보통 *cls* 의 인스턴스).

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

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 같다.

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

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

`object.`**`__init__`**`(`*self*`[, ...])`
Called after the instance has been created (by *__new__()*), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an *__init__()* method, the derived class's *__init__()* method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `BaseClass.__init__(self, [args...])`.

Because *__new__()* and *__init__()* work together in constructing objects (*__new__()* to create it, and *__init__()* to customise it), no non-`None` value may be returned by *__init__()*; doing so will cause a `TypeError` to be raised at runtime.

`object.`**`__del__`**`(`*self*`)`
Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a *__del__()* method, the derived class's *__del__()* method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended!) for the *__del__()* method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that *__del__()* methods are called for objects that still exist when the interpreter exits.

참고: `del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include: circular references between objects (e.g., a doubly-linked list or a tree data structure with parent and child pointers); a reference to the object on the stack frame of a function that caught an exception (the traceback stored in `sys.exc_traceback` keeps the stack frame alive); or a reference to the object on the stack frame that raised an unhandled exception in interactive mode (the traceback stored in `sys.last_traceback` keeps the stack frame alive). The first situation can only be remedied by explicitly breaking the cycles; the latter two situations can be resolved by storing `None` in `sys.exc_traceback` or `sys.last_traceback`. Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level *__del__()* methods involved. Refer to the documentation for the `gc` module for more information about how *__del__()* methods are handled by the cycle detector, particularly the description of the `garbage` value.

> **경고:** Due to the precarious circumstances under which *__del__()* methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. Also, when *__del__()* is invoked in response to a module being deleted (e.g., when execution of the program is done), other globals referenced by the *__del__()* method may already have been deleted or in the process of being torn down (e.g. the import machinery shutting down). For this reason, *__del__()* methods should do the absolute minimum needed to maintain external invariants. Starting with version 1.5, Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the *__del__()* method is called.

See also the `-R` command-line option.

`object.`**`__repr__`**`(`*self*`)`
Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the《official》string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object. If a class defines *__repr__()* but not *__str__()*, then *__repr__()* is also used when an《informal》string representation of instances of that class is required.

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

`object.`**`__str__`**`(`*self*`)`
Called by the `str()` built-in function and by the *print* statement to compute the《informal》string representation of an object. This differs from *__repr__()* in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

`object.`**`__lt__`**`(`*self*`,` *other*`)`
`object.`**`__le__`**`(`*self*`,` *other*`)`
`object.`**`__eq__`**`(`*self*`,` *other*`)`
`object.`**`__ne__`**`(`*self*`,` *other*`)`
`object.`**`__gt__`**`(`*self*`,` *other*`)`
`object.`**`__ge__`**`(`*self*`,` *other*`)`
버전 2.1에 추가.

These are the so-called《rich comparison》methods, and are called for comparison operators in preference to *__cmp__()* below. The correspondence between operator symbols and method names is as follows: `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` and `x<>y` call `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`.

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

There are no implied relationships among the comparison operators. The truth of `x==y` does not imply that `x!=y` is false. Accordingly, when defining *__eq__()*, one should also define *__ne__()* so that the operators will behave as expected. See the paragraph on *__hash__()* for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, *__lt__()* and *__gt__()* are each other's reflection, *__le__()* and *__ge__()* are each other's reflection, and *__eq__()* and *__ne__()* are their own reflection.

Arguments to rich comparison methods are never coerced.

To automatically generate ordering operations from a single root operation, see `functools.total_ordering()`.

`object.__cmp__(self, other)`
Called by comparison operations if rich comparison (see above) is not defined. Should return a negative integer if `self < other`, zero if `self == other`, a positive integer if `self > other`. If no *__cmp__()*, *__eq__()* or *__ne__()* operation is defined, class instances are compared by object identity (《address》). See also the description of *__hash__()* for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys. (Note: the restriction that exceptions are not propagated by *__cmp__()* has been removed since Python 1.5.)

`object.__rcmp__(self, other)`
버전 2.1에서 변경: No longer supported.

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

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

If a class does not define a *__cmp__()* or *__eq__()* method it should not define a *__hash__()* operation either; if it defines *__cmp__()* or *__eq__()* but not *__hash__()*, its instances will not be usable in hashed collections. If a class defines mutable objects and implements a *__cmp__()* or *__eq__()* method, it should not implement *__hash__()*, since hashable collection implementations require that an object's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

User-defined classes have *__cmp__()* and *__hash__()* methods by default; with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns a result derived from `id(x)`.

Classes which inherit a *__hash__()* method from a parent class but change the meaning of *__cmp__()* or *__eq__()* such that the hash value returned is no longer appropriate (e.g. by switching to a value-based concept of equality instead of the default identity based equality) can explicitly flag themselves as being unhashable by setting `__hash__ = None` in the class definition. Doing so means that not only will instances of the class raise an appropriate `TypeError` when a program attempts to retrieve their hash value, but they will also be correctly identified as unhashable when checking `isinstance(obj, collections.Hashable)` (unlike classes which define their own *__hash__()* to explicitly raise `TypeError`).

버전 2.5에서 변경: *__hash__()* may now also return a long integer object; the 32-bit integer is then derived from the hash of that object.

버전 2.6에서 변경: *__hash__* may now be set to `None` to explicitly flag instances of a class as unhashable.

`object.__nonzero__(self)`
Called to implement truth value testing and the built-in operation `bool()`; should return `False` or `True`, or their integer equivalents `0` or `1`. When this method is not defined, *__len__()* is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither *__len__()* nor *__nonzero__()*, all its instances are considered true.

`object.__unicode__(self)`
Called to implement `unicode()` built-in; should return a Unicode object. When this method is not defined, string conversion is attempted, and the result of string conversion is converted to Unicode using the system default encoding.

3.4.2 어트리뷰트 액세스 커스터마이제이션

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

`object.`**`__getattr__`**(*self*, *name*)
Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for `self`). `name` is the attribute name. This method should return the (computed) attribute value or raise an `AttributeError` exception.

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

`object.`**`__setattr__`**(*self*, *name*, *value*)
Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). *name* is the attribute name, *value* is the value to be assigned to it.

If *__setattr__()* wants to assign to an instance attribute, it should not simply execute `self.name = value` — this would cause a recursive call to itself. Instead, it should insert the value in the dictionary of instance attributes, e.g., `self.__dict__[name] = value`. For new-style classes, rather than accessing the instance dictionary, it should call the base class method with the same name, for example, `object.__setattr__(self, name, value)`.

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

More attribute access for new-style classes

The following methods only apply to new-style classes.

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

참고: This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See *Special method lookup for new-style classes*.

디스크립터 구현하기

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

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

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

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

디스크립터 호출하기

일반적으로, 디스크립터는 《결합한 동작(binding behavior)》을 가진 객체 어트리뷰트다. 어트리뷰트 액세스가 디스크립터 프로토콜(descriptor protocol)의 메서드들에 의해 재정의된다: *`__get__()`*, *`__set__()`*, *`__delete__()`*. 이 메서드들 중 하나라도 정의되어 있으면, 디스크립터라고 부른다.

어트리뷰트 액세스의 기본 동작은 객체의 딕셔너리에서 어트리뷰트를 읽고, 쓰고, 삭제하는 것이다. 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 구성된다.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called. Note that descriptors are only invoked for new style objects or classes (ones that subclass `object()` or `type()`).

디스크립터 호출의 시작점은 결합(binding)이다, `a.x`. 어떻데 인자들이 조합되는지는 `a` 에 따라 다르다:

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

인스턴스 결합 If binding to a new-style object instance, `a.x` is transformed into the call: `type(a).__dict__['x'].__get__(a, type(a))`.

클래스 결합 If binding to a new-style class, `A.x` is transformed into the call: `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 호출한다: `A.__dict__['m'].__get__(obj, obj.__class__)`.

인스턴스 결합의 경우, 디스크립터 호출의 우선순위는 어떤 디스크립터 메서드가 정의되어있는지에 따라 다르다. 디스크립터는 *`__get__()`*, *`__set__()`*, *`__delete__()`* 를 어떤 조합으로도 정의할 수 있다. 만약 *`__get__()`* 를 정의하지 않는다면, 어트리뷰트 액세스는, 객체의 인스턴스 딕셔너리에 값이 있지 않은 이상 디스크립터 객체 자신을 돌려준다. 만약 디스크립터가 *`__set__()`* 이나 *`__delete__()`* 중 어느 하나나 둘 다 정의하면, 데이터 디스크립터(data descriptor)다. 둘 다 정의하지 않는다면 비데이터 디스크립터다(non-data descriptor). 보통, 데이터 디스크립터가 *`__get__()`* 과 *`__set__()`* 을 모두 정의하는 반면, 비데이터 디스크립터는 *`__get__()`* 메서드만 정의한다. *`__set__()`* 과 *`__get__()`* 이 있는 데이터 디스크립터는 이스턴스 딕셔너리에 있는 값에 우선한다. 반면에 비데이터 디스크립터는 인스턴스보다 우선순위가 낮다.

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

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

__slots__

By default, instances of both old and new-style classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.

The default can be overridden by defining *__slots__* in a new-style class definition. The *__slots__* declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because *__dict__* is not created for each instance.

__slots__

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

버전 2.2에 추가.

__slots__ 사용에 관한 노트

- When inheriting from a class without *__slots__*, the *__dict__* attribute of that class will always be accessible, so a *__slots__* definition in the subclass is meaningless.
- *__dict__* 변수가 없으므로 인스턴스는 *__slots__* 정의에 나열되지 않은 새 변수를 대입할 수 없다. 나열되지 않은 변수명으로 대입하려고 하면 `AttributeError` 를 일으킨다. 만약 동적으로 새 변수를 대입하는 것이 필요하다면, *__slots__* 선언의 문자열 시퀀스에 `'__dict__'` 를 추가한다.

  버전 2.3에서 변경: Previously, adding `'__dict__'` to the *__slots__* declaration would not enable the assignment of new attributes not specifically listed in the sequence of instance variable names.
- 인스턴스마다 *__weakref__* 변수가 없으므로, *__slots__* 를 정의하는 클래스는 인스턴스에 대한 약한 참조 (weak reference) 를 지원하지 않는다. 만약 약한 참조 지원이 필요하다면, *__slots__* 선언의 문자열 시퀀스에 `'__weakref__'` 를 추가한다.

  버전 2.3에서 변경: Previously, adding `'__weakref__'` to the *__slots__* declaration would not enable support for weak references.
- *__slots__* 는 각 변수 이름마다 디스크립터를 만드는 방식으로 클래스 수준에서 구현된다(디스크립터 구현하기). 결과적으로, 클래스 어트리뷰트는 *__slots__* 로 정의된 인스턴스 변수들을 위한 기본값을 제공할 목적으로 사용될 수 없다. 클래스 어트리뷰트는 디스크립터 대입을 무효로 한다.
- The action of a *__slots__* declaration is limited to the class where it is defined. As a result, subclasses will have a *__dict__* unless they also define *__slots__* (which must only contain names of any *additional* slots).
- 클래스가 베이스 클래스의 *__slots__* 에 정의된 이름과 같은 이름의 변수를 *__slots__* 에 선언한다면, 베이스 클래스가 정의한 변수는 액세스할 수 없는 상태가 된다(베이스 클래스로부터 디스크립터를 직접 조회하는 경우는 예외다). 이것은 프로그램을 정의되지 않은 상태로 보내게 된다. 미래에는, 이를 방지하기 위한 검사가 추가될 것이다.
- Nonempty *__slots__* does not work for classes derived from《variable-length》built-in types such as `long`, `str` and `tuple`.
- *__slots__* 에는 문자열 이외의 이터러블을 대입할 수 있다. 매핑도 역시 사용할 수 있다. 하지만, 미래에, 각 키에 대응하는 값들의 의미가 부여될 수 있다.

- 두 클래스가 같은 *__slots__* 을 갖는 경우만 *__class__* 대입이 동작한다.

  버전 2.6에서 변경: Previously, *__class__* assignment raised an error if either new or old class had *__slots__*.

3.4.3 클래스 생성 커스터마이제이션

By default, new-style classes are constructed using `type()`. A class definition is read into a separate namespace and the value of class name is bound to the result of `type(name, bases, dict)`.

When the class definition is read, if *__metaclass__* is defined then the callable assigned to it will be called instead of `type()`. This allows classes or functions to be written which monitor or alter the class creation process:

- Modifying the class dictionary prior to the class being created.
- Returning an instance of another class – essentially performing the role of a factory function.

These steps will have to be performed in the metaclass's *__new__()* method – `type.__new__()` can then be called from this method to create a class with different properties. This example adds a new element to the class dictionary before creating the class:

```
class metacls(type):
    def __new__(mcs, name, bases, dict):
        dict['foo'] = 'metacls was here'
        return type.__new__(mcs, name, bases, dict)
```

You can of course also override other class methods (or add new methods); for example defining a custom *__call__()* method in the metaclass allows custom behavior when the class is called, e.g. not always creating a new instance.

__metaclass__
: This variable can be any callable accepting arguments for `name`, `bases`, and `dict`. Upon class creation, the callable is used instead of the built-in `type()`.

  버전 2.2에 추가.

The appropriate metaclass is determined by the following precedence rules:

- If `dict['__metaclass__']` exists, it is used.
- Otherwise, if there is at least one base class, its metaclass is used (this looks for a *__class__* attribute first and if not found, uses its type).
- Otherwise, if a global variable named __metaclass__ exists, it is used.
- Otherwise, the old-style, classic metaclass (types.ClassType) is used.

The potential uses for metaclasses are boundless. Some ideas that have been explored including logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.4.4 인스턴스 및 서브 클래스 검사 커스터마이제이션

버전 2.6에 추가.

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

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

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

class.**__subclasscheck__**(*self*, *subclass*)
subclass 가 (직접적이거나 간접적으로) *class* 의 서브 클래스로 취급될 수 있으면 참을 돌려준다. 만약 정의되면, issubclass(subclass, class) 를 구현하기 위해 호출된다.

이 메서드들은 클래스의 형 (메타 클래스) 에서 조회된다는 것에 주의해야 한다. 실제 클래스에서 클래스 메서드로 정의될 수 없다. 이것은 인스턴스에 대해 호출되는 특수 메서드들의 조회와 일관성 있다. 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 보라) 를 추가하고자 하는 데 있다.

3.4.5 콜러블 객체 흉내 내기

object.**__call__**(*self*[, *args...*])
인스턴스가 함수처럼《호출될》때 호출된다; 이 메서드가 정의되면, x(arg1, arg2, ...) 는 x.__call__(arg1, arg2, ...) 의 줄인 표현이다.

3.4.6 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. (For backwards compatibility, the method *__getslice__()* (see below) can also be defined to handle simple, but not extended slices.) It is also recommended that mappings provide the methods keys(), values(), items(), has_key(), get(), clear(), setdefault(), iterkeys(), itervalues(), iteritems(), pop(), popitem(), copy(), and update() behaving similar to those for Python's standard dictionary objects. The UserDict module provides a DictMixin 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 *__coerce__()* or 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 be equivalent of has_key(); 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 be the same as iterkeys(); for sequences, it should iterate through the values.

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

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

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

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

---

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

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

object.**__missing__**(*self*, *key*)
dict.*__getitem__()* 이 dict 서브 클래스에서 키가 딕셔너리에 없으면 self[key] 를 구현하기 위해 호출한다.

object.**__iter__**(*self*)
This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container, and should also be made available as the method iterkeys().

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

object.**__reversed__**(*self*)
reversed() 내장 함수가 역 이터레이션(reverse iteration)을 구현하기 위해 (있다면) 호출한다. 컨테이너에 있는 객체들을 역 순으로 탐색하는 새 이터레이터 객체를 돌려줘야 한다.

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

버전 2.6에 추가.

멤버십 검사 연산자들(*in* 과 *not in*) 은 보통 시퀀스에 대한 이터레이션으로 구현된다. 하지만, 컨테이너 객체는 더 효율적인 구현을 다음과 같은 특수 메서드를 통해 제공할 수 있다. 이 경우 객체는 시퀀스일 필요도 없다.

object.**__contains__**(*self*, *item*)
멤버십 검사 연산자를 구현하기 위해 호출된다. *item* 이 *self* 에 있으면 참을, 그렇지 않으면 거짓을 돌려줘야 한다. 매핑 객체의 경우, 키-값 쌍이 아니라 매핑의 키가 고려되어야 한다.

__contains__() 를 정의하지 않는 객체의 경우, 멤버십 검사는 먼저 *__iter__()* 를 통한 이터레이션을 시도한 후, *__getitem__()* 을 통한 낡은 시퀀스 이터레이션 프로토콜을 시도한다. *membership-test-details* 섹션을 참고하라.

3.4.7 Additional methods for emulation of sequence types

The following optional methods can be defined to further emulate sequence objects. Immutable sequences methods should at most only define *__getslice__()*; mutable sequences might define all three methods.

`object.`**`__getslice__`**(*self*, *i*, *j*)
버전 2.0부터 폐지: Support slice objects as parameters to the *__getitem__()* method. (However, built-in types in CPython currently still implement *__getslice__()*. Therefore, you have to override it in derived classes when implementing slicing.)

Called to implement evaluation of `self[i:j]`. The returned object should be of the same type as *self*. Note that missing *i* or *j* in the slice expression are replaced by zero or `sys.maxsize`, respectively. If negative indexes are used in the slice, the length of the sequence is added to that index. If the instance does not implement the *__len__()* method, an `AttributeError` is raised. No guarantee is made that indexes adjusted this way are not still negative. Indexes which are greater than the length of the sequence are not modified. If no *__getslice__()* is found, a slice object is created instead, and passed to *__getitem__()* instead.

`object.`**`__setslice__`**(*self*, *i*, *j*, *sequence*)
Called to implement assignment to `self[i:j]`. Same notes for *i* and *j* as for *__getslice__()*.

This method is deprecated. If no *__setslice__()* is found, or for extended slicing of the form `self[i:j:k]`, a slice object is created, and passed to *__setitem__()*, instead of *__setslice__()* being called.

`object.`**`__delslice__`**(*self*, *i*, *j*)
Called to implement deletion of `self[i:j]`. Same notes for *i* and *j* as for *__getslice__()*. This method is deprecated. If no *__delslice__()* is found, or for extended slicing of the form `self[i:j:k]`, a slice object is created, and passed to *__delitem__()*, instead of *__delslice__()* being called.

Notice that these methods are only invoked when a single slice with a single colon is used, and the slice method is available. For slice operations involving extended slice notation, or in absence of the slice methods, *__getitem__()*, *__setitem__()* or *__delitem__()* is called with a slice object as argument.

The following example demonstrate how to make your program or module compatible with earlier versions of Python (assuming that methods *__getitem__()*, *__setitem__()* and *__delitem__()* support slice objects as arguments):

```
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final

        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note the calls to `max()`; these are necessary because of the handling of negative indices before the `__*slice__()` methods are called. When negative indexes are used, the `__*item__()` methods receive them as provided, but the

`__*slice__()` methods get a《cooked》form of the index values. For each negative index value, the length of the sequence is added to the index before calling the method (which may still result in a negative index); this is the customary handling of negative indexes by the built-in sequence types, and the `__*item__()` methods are expected to do this as well. However, since they should already be doing that, negative indexes cannot be passed in; they must be constrained to the bounds of the sequence before being passed to the `__*item__()` methods. Calling `max(0, i)` conveniently returns the proper value.

3.4.8 숫자 형 흉내 내기

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

`object.`**`__add__`**(*self*, *other*)
`object.`**`__sub__`**(*self*, *other*)
`object.`**`__mul__`**(*self*, *other*)
`object.`**`__floordiv__`**(*self*, *other*)
`object.`**`__mod__`**(*self*, *other*)
`object.`**`__divmod__`**(*self*, *other*)
`object.`**`__pow__`**(*self*, *other*[, *modulo*])
`object.`**`__lshift__`**(*self*, *other*)
`object.`**`__rshift__`**(*self*, *other*)
`object.`**`__and__`**(*self*, *other*)
`object.`**`__xor__`**(*self*, *other*)
`object.`**`__or__`**(*self*, *other*)

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

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

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

The division operator (`/`) is implemented by these methods. The *`__truediv__()`* method is used when `__future__.division` is in effect, otherwise *`__div__()`* is used. If only one of these two methods is defined, the object will not support division in the alternate context; `TypeError` will be raised instead.

`object.`**`__radd__`**(*self*, *other*)
`object.`**`__rsub__`**(*self*, *other*)
`object.`**`__rmul__`**(*self*, *other*)
`object.`**`__rdiv__`**(*self*, *other*)
`object.`**`__rtruediv__`**(*self*, *other*)
`object.`**`__rfloordiv__`**(*self*, *other*)
`object.`**`__rmod__`**(*self*, *other*)
`object.`**`__rdivmod__`**(*self*, *other*)
`object.`**`__rpow__`**(*self*, *other*)
`object.`**`__rlshift__`**(*self*, *other*)
`object.`**`__rrshift__`**(*self*, *other*)
`object.`**`__rand__`**(*self*, *other*)
`object.`**`__rxor__`**(*self*, *other*)

object.**__ror__**(*self*, *other*)
These methods are called to implement the binary arithmetic operations (+, -, *, /, %, divmod(), pow(), **, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation and the operands are of different types.[2] For instance, to evaluate the expression x - y, where *y* is an instance of a class that has an *__rsub__()* method, y.__rsub__(x) is called if x.__sub__(y) returns *NotImplemented*.

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

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 호출된다. 이 동작은 서브 클래스가 조상들의 연산을 재정의할 수 있도록 한다.

---

object.**__iadd__**(*self*, *other*)
object.**__isub__**(*self*, *other*)
object.**__imul__**(*self*, *other*)
object.**__idiv__**(*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*)
These methods are called to implement the augmented arithmetic assignments (+=, -=, *=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, the augmented assignment falls back to the normal methods. For instance, to execute the statement x += y, where *x* is an instance of a class that has an *__iadd__()* method, x.__iadd__(y) is called. If *x* is an instance of a class that does not define a *__iadd__()* method, x.__add__(y) and y.__radd__(x) are considered, as with the evaluation of x + y.

object.**__neg__**(*self*)
object.**__pos__**(*self*)
object.**__abs__**(*self*)
object.**__invert__**(*self*)
일 항 산술 연산(-, +, abs(), ~)을 구현하기 위해 호출된다.

object.**__complex__**(*self*)
object.**__int__**(*self*)
object.**__long__**(*self*)
object.**__float__**(*self*)
Called to implement the built-in functions complex(), int(), long(), and float(). Should return a value of the appropriate type.

object.**__oct__**(*self*)
object.**__hex__**(*self*)
Called to implement the built-in functions oct() and hex(). Should return a string value.

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

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

Called to implement `operator.index()`. Also called whenever Python needs an integer object (such as in slicing). Must return an integer (int or long).

버전 2.5에 추가.

`object.__coerce__`(*self*, *other*)
Called to implement 《mixed-mode》 numeric arithmetic. Should either return a 2-tuple containing *self* and *other* converted to a common numeric type, or `None` if conversion is impossible. When the common type would be the type of `other`, it is sufficient to return `None`, since the interpreter will also ask the other object to attempt a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to the other type here). A return value of `NotImplemented` is equivalent to returning `None`.

3.4.9 Coercion rules

This section used to document the rules for coercion. As the language has evolved, the coercion rules have become hard to document precisely; documenting what one version of one particular implementation does is undesirable. Instead, here are some informal guidelines regarding coercion. In Python 3, coercion will not be supported.

- If the left operand of a % operator is a string or Unicode object, no coercion takes place and the string formatting operation is invoked instead.
- It is no longer recommended to define a coercion operation. Mixed-mode operations on types that don't define coercion pass the original arguments to the operation.
- New-style classes (those derived from `object`) never invoke the *__coerce__()* method in response to a binary operator; the only time *__coerce__()* is invoked is when the built-in function `coerce()` is called.
- For most intents and purposes, an operator that returns `NotImplemented` is treated the same as one that is not implemented at all.
- Below, `__op__()` and `__rop__()` are used to signify the generic method names corresponding to an operator; `__iop__()` is used for the corresponding in-place operator. For example, for the operator 〈+〉, *__add__()* and *__radd__()* are used for the left and right variant of the binary operator, and *__iadd__()* for the in-place variant.
- For objects *x* and *y*, first `x.__op__(y)` is tried. If this is not implemented or returns `NotImplemented`, `y.__rop__(x)` is tried. If this is also not implemented or returns `NotImplemented`, a `TypeError` exception is raised. But see the following exception:
- Exception to the previous item: if the left operand is an instance of a built-in type or a new-style class, and the right operand is an instance of a proper subclass of that type or class and overrides the base's `__rop__()` method, the right operand's `__rop__()` method is tried *before* the left operand's `__op__()` method.

  This is done so that a subclass can completely override binary operators. Otherwise, the left operand's `__op__()` method would always accept the right operand: when an instance of a given class is expected, an instance of a subclass of that class is always acceptable.
- When either operand type defines a coercion, this coercion is called before that type's `__op__()` or `__rop__()` method is called, but no sooner. If the coercion returns an object of a different type for the operand whose coercion is invoked, part of the process is redone using the new object.
- When an in-place operator (like 〈+=〉) is used, if the left operand implements `__iop__()`, it is invoked without any coercion. When the operation falls back to `__op__()` and/or `__rop__()`, the normal coercion rules apply.
- In `x + y`, if *x* is a sequence that implements sequence concatenation, sequence concatenation is invoked.
- In `x * y`, if one operand is a sequence that implements sequence repetition, and the other is an integer (`int` or `long`), sequence repetition is invoked.

- Rich comparisons (implemented by methods *__eq__()* and so on) never use coercion. Three-way comparison (implemented by *__cmp__()*) does use coercion under the same conditions as other binary operations use it.
- In the current implementation, the built-in numeric types `int`, `long`, `float`, and `complex` do not use coercion. All these types implement a *__coerce__()* method, for use by the built-in `coerce()` function.

  버전 2.7에서 변경: The complex type no longer makes implicit calls to the *__coerce__()* method for mixed-type binary arithmetic operations.

3.4.10 with 문 컨텍스트 관리자

버전 2.5에 추가.

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

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 있다.

컨텍스트 관리자에 대한 더 자세한 정보는 typecontextmanager 에 나온다.

`object.`**`__enter__`**`(self)`
: 이 객체와 연관된 실행시간 컨텍스트에 진입한다. *with* 문은 *as* 절로 지정된 대상이 있다면, 이 메서드의 반환 값을 연결한다.

`object.`**`__exit__`**`(self, exc_type, exc_value, traceback)`
: 이 객체와 연관된 실행시간 컨텍스트를 종료한다. 파라미터들은 컨텍스트에서 벗어나게 만든 예외를 기술한다. 만약 컨텍스트가 예외 없이 종료한다면, 세 인자 모두 `None` 이 된다.

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

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

더 보기:

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

3.4.11 Special method lookup for old-style classes

For old-style classes, special methods are always looked up in exactly the same way as any other method or attribute. This is the case regardless of whether the method is being looked up explicitly as in `x.__getitem__(i)` or implicitly as in `x[i]`.

This behaviour means that special methods may exhibit different behaviour for different instances of a single old-style class if the appropriate special attributes are set differently:

```
>>> class C:
...     pass
...
>>> c1 = C()
>>> c2 = C()
>>> c1.__len__ = lambda: 5
>>> c2.__len__ = lambda: 9
>>> len(c1)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
5
>>> len(c2)
9
```

3.4.12 Special method lookup for new-style classes

For new-style classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception (unlike the equivalent example with old-style classes):

```
>>> class C(object):
...     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)들에 의해 구현되는 *__hash__()* 나 *__repr__()* 과 같은 많은 특수 메서드들과 관련이 있다. 만약 이 메서드들에 대한 묵시적인 조회가 일반적인 조회 프로세스를 거친다면, 형 객체 자체에 대해 호출되었을 때 실패하게 된다:

```
>>> 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 있다.

```
>>> type(1).__hash__(1) == hash(1)
True
>>> 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 우회한다:

```
>>> 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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> 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 한다), 인터프리터 내부에서의 속도 최적화를 위한 상당한 기회를 제공한다.

CHAPTER 4

실행 모델

4.1 이름과 연결(binding)

Names refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the *binding* of that name established in the innermost function block containing the use.

A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command specified on the interpreter command line with the 〈**-c**〉 option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and to the *exec* statement is a code block. The expression read and evaluated by the built-in function `input()` is a code block.

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

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes generator expressions since they are implemented using a function scope. This means that the following will fail:

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

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

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

When a name is not found at all, a `NameError` exception is raised. If the name refers to a local variable that has not been bound, a `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

The following constructs bind names: formal parameters to functions, *import* statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, *for* loop header, in the second position of an *except* clause header or after *as* in a *with* statement. The *import* statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a *del* statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is referenced by an enclosing scope; the compiler will report a `SyntaxError`.

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

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

If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module `__builtin__`. The global namespace is searched first. If the name is not found there, the builtins namespace is searched. The global statement must precede all uses of the name.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `__builtin__` (note: no ⟨s⟩); when in any other module, `__builtins__` is an alias for the dictionary of the `__builtin__` module itself. `__builtins__` can be set to a user-created dictionary to create a weak form of restricted execution.

CPython implementation detail: Users should not touch `__builtins__`; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should *import* the `__builtin__` (no ⟨s⟩) module and modify its attributes appropriately.

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

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

A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution. The namespace of the class definition becomes the attribute dictionary of the class. Names defined at the class scope are not visible in methods.

4.1.1 동적 기능과의 상호작용

There are several cases where Python statements are illegal when used in conjunction with nested scopes that contain free variables.

If a variable is referenced in an enclosing scope, it is illegal to delete the name. An error will be reported at compile time.

If the wild card form of import — `import *` — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError`.

If *exec* is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError` unless the exec explicitly specifies the local namespace for the *exec*. (In other words, `exec obj` would be illegal, but `exec obj in ns` would be legal.)

The `eval()`, `execfile()`, and `input()` functions and the *exec* statement do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace.[1] The *exec* statement and the `eval()` and `execfile()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.2 예외

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

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

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

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is `SystemExit`.

예외는 클래스 인스턴스로 구분된다. *except* 절은 인스턴스의 클래스에 따라 선택된다: 인스턴스의 클래스나 그것의 베이스 클래스를 가리켜야 한다. 인스턴스는 핸들러가 수신할 수 있고 예외적인 조건에 대한 추가적인 정보를 포함할 수 있다.

Exceptions can also be identified by strings, in which case the *except* clause is selected by object identity. An arbitrary value can be raised along with the identifying string which can be passed to the handler.

참고: Messages to exceptions are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

섹션 *try* 문 에서 *try* 문, *raise* 문 에서 *raise* 문에 대한 설명이 제공된다.

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

CHAPTER 5

표현식

이 장은 파이썬에서 사용되는 표현식 요소들의 의미를 설명한다.

문법 유의 사항: 여기와 이어지는 장에서는, 구문 분석이 아니라 문법을 설명하기 위해 확장 BNF 표기법을 사용한다. 문법 규칙이 다음과 같은 형태를 가지고,

```
name   ::=   othername
```

뜻(semantics)을 주지 않으면, 이 형태의 `name` 의 뜻은 `othername` 과 같다.

5.1 산술 변환

When a description of an arithmetic operator below uses the phrase《the numeric arguments are converted to a common type,》the arguments are coerced using the coercion rules listed at *Coercion rules*. If both arguments are standard numeric types, the following coercions are applied:

- 어느 한 인자가 복소수면 다른 하나는 복소수로 변환된다;
- 그렇지 않고, 어느 한 인자가 실수면, 다른 하나는 실수로 변환된다;
- otherwise, if either argument is a long integer, the other is converted to long integer;
- otherwise, both must be plain integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string left argument to the〈%〉operator). Extensions can define their own coercions.

5.2 아톰 (Atoms)

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom      ::=  identifier | literal | enclosure
enclosure ::=  parenth_form | list_display
               | generator_expression | dict_display | set_display
               | string_conversion | yield_atom
```

5.2.1 식별자 (이름)

아톰으로 등장하는 식별자는 이름이다. 구문 분석에 대해서는 식별자와 키워드 섹션을, 이름과 연결에 대한 문서는 이름과 연결 *(binding)* 섹션을 보면 된다.

이름이 객체에 연결될 때, 아톰의 값을 구하면 객체가 나온다. 이름이 연결되지 않았을 때, 값을 구하려고 하면 `NameError` 예외가 일어난다.

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 간주한다. 비공개 이름은 그 들을 위한 코드가 만들어지기 전에 더 긴 형태로 변환된다. 이 변환은 그 이름의 앞에 클래스 이름을 삽입하는데, 클래스 이름의 처음에 오는 모든 밑줄을 제거한 후, 하나의 밑줄을 추가한다. 예를 들어, `Ham` 이라는 이름의 클래스에 식별자 `__spam` 이 등장하면, `_Ham__spam` 으로 변환된다. 이 변환은 식별자가 사용되는 문법적인 문맥에 무관하다. 변환된 이름이 극단적으로 길면 (255 자보다 길면), 구현이 정의한 잘라내기가 발생할 수 있다. 클래스 이름이 밑줄로만 구성되어 있으면, 변환은 일어나지 않는다.

5.2.2 리터럴 (Literals)

Python supports string literals and various numeric literals:

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

Evaluation of a literal yields an object of the given type (string, integer, long integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section 리터럴 for details.

모든 리터럴은 불변 데이터형에 대응하기 때문에, 객체의 아이덴티티는 값 보다 덜 중요하다. 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 있다.

5.2.3 괄호 안에 넣은 형

괄호 안에 넣은 형은, 괄호로 둘러싸인 생략 가능한 표현식 목록이다:

```
parenth_form  ::=  "(" [expression_list] ")"
```

괄호 안에 넣은 표현식 목록은, 무엇이건 그 표현식 목록이 산출하는 것이 된다: 목록이 적어도 하나의 쉼표를 포함하면, 튜플이 된다; 그렇지 않으면 표현식 목록을 구성한 단일 표현식이 된다.

빈 괄호 쌍은 빈 튜플 객체를 만든다. 튜플은 불변이기 때문에 리터럴의 규칙이 적용된다(즉, 두 개의 빈 튜플은 같은 객체일 수도 있고 그렇지 않을 수도 있다).

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

5.2.4 리스트 디스플레이

리스트 디스플레이는 꺾쇠괄호(square brackets)로 둘러싸인 표현식의 나열인데 비어있을 수 있다:

```
list_display          ::=  "[" [expression_list | list_comprehension] "]"
list_comprehension    ::=  expression list_for
list_for              ::=  "for" target_list "in" old_expression_list [list_iter]
old_expression_list   ::=  old_expression [("," old_expression)+ [","]]
old_expression        ::=  or_test | old_lambda_expr
list_iter             ::=  list_for | list_if
list_if               ::=  "if" old_expression [list_iter]
```

A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a list comprehension is supplied, it consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new list are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached[1].

5.2.5 Displays for sets and dictionaries

For constructing a set or a dictionary Python provides special syntax called《displays》, each of them in two flavors:

- 컨테이너의 내용을 명시적으로 나열하거나,
- 일련의 루프와 필터링 지시들을 통해 계산되는데, 컴프리헨션 *(comprehension)* 이라고 불린다.

컴프리헨션의 공통 문법 요소들은 이렇다:

```
comprehension  ::=  expression comp_for
comp_for       ::=  "for" target_list "in" or_test [comp_iter]
comp_iter      ::=  comp_for | comp_if
comp_if        ::=  "if" expression_nocond [comp_iter]
```

[1] In Python 2.3 and later releases, a list comprehension《leaks》the control variables of each `for` it contains into the containing scope. However, this behavior is deprecated, and relying on it will not work in Python 3.

컴프리헨션은 하나의 표현식과 그 뒤를 따르는 최소한 하나의 *for* 절과 없거나 여러 개의 *for* 또는 *if* 절로 구성된다. 이 경우, 새 컨테이너의 요소들은 각 *for* 또는 *if* 절이 왼쪽에서 오른쪽으로 중첩된 블록을 이루고, 가장 안쪽에 있는 블록에서 표현식의 값을 구해서 만들어낸 것들이다.

Note that the comprehension is executed in a separate scope, so names assigned to in the target list don't《leak》in the enclosing scope.

5.2.6 제너레이터 표현식 (Generator expressions)

제너레이터 표현식은 괄호로 둘러싸인 간결한 제너레이터 표기법이다.

```
generator_expression  ::=  "(" expression comp_for ")"
```

제너레이터 표현식은 새 제너레이터 객체를 만든다. 문법은 꺾쇠괄호나 중괄호 대신 괄호로 둘러싸인다는 점만 제외하면 컴프리헨션과 같다.

Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for generator object (in the same fashion as normal generators). However, the leftmost *for* clause is immediately evaluated, so that an error produced by it can be seen before any other possible error in the code that handles the generator expression. Subsequent *for* clauses cannot be evaluated immediately since they may depend on the previous *for* loop. For example: `(x*y for x in range(10) for y in bar(x))`.

The parentheses can be omitted on calls with only one argument. See section 호출 for the detail.

5.2.7 딕셔너리 디스플레이

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

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

딕셔너리 디스플레이는 새 딕셔너리 객체를 만든다.

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

딕셔너리 컴프리헨션은, 리스트와 집합 컴프리헨션에 대비해서, 일반적인《for》와《if》절 앞에 콜론으로 분리된 두 개의 표현식을 필요로 한다. 컴프리헨션이 실행될 때, 만들어지는 키와 값 요소들이 만들어지는 순서대로 딕셔너리에 삽입된다.

킷값의 형에 대한 제약은 앞의 섹션 표준형 계층 에서 나열되었다. (요약하자면, 키 형은 해시 가능 해야 하는데, 모든 가변 객체들이 제외된다.) 중복된 키 간의 충돌은 감지되지 않는다; 주어진 키에 대해 저장된 마지막 (구문상으로 디스플레이의 가장 오른쪽에 있는) 데이터가 우선한다.

5.2.8 집합 디스플레이

집합 디스플레이는 중괄호(curly braces)로 표시되고, 키와 값을 분리하는 콜론(colon)이 없는 것으로 딕셔너리 디스플레이와 구분될 수 있다.

```
set_display  ::=  "{" (expression_list | comprehension) "}"
```

집합 디스플레이는 새 가변 집합 객체를 만드는데, 그 내용은 표현식의 시퀀스나 컴프리헨션으로 지정된다. 쉼표로 분리된 표현식의 목록이 제공될 때, 그 요소들은 왼쪽에서 오른쪽으로 값이 구해지고, 집합 객체에 더해진다. 컴프리헨션이 제공될 때, 집합은 컴프리헨션으로 만들어지는 요소들로 구성된다.

빈 집합은 `{}` 으로 만들어질 수 없다; 이 리터럴은 빈 딕셔너리를 만든다.

5.2.9 String conversions

A string conversion is an expression list enclosed in reverse (a.k.a. backward) quotes:

```
string_conversion  ::=  "`" expression_list "`"
```

A string conversion evaluates the contained expression list and converts the resulting object into a string according to rules specific to its type.

If the object is a string, a number, `None`, or a tuple, list or dictionary containing only objects whose type is one of these, the resulting string is a valid Python expression which can be passed to the built-in function `eval()` to yield an expression with the same value (or an approximation, if floating point numbers are involved).

(In particular, converting a string adds quotes around it and converts《funny》characters to escape sequences that are safe to print.)

Recursive objects (for example, lists or dictionaries that contain a reference to themselves, directly or indirectly) use `...` to indicate a recursive reference, and the result cannot be passed to `eval()` to get an equal value (`SyntaxError` will be raised instead).

The built-in function `repr()` performs exactly the same conversion in its argument as enclosing it in parentheses and reverse quotes does. The built-in function `str()` performs a similar but more user-friendly conversion.

5.2.10 일드 표현식(Yield expressions)

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

버전 2.5에 추가.

The *yield* expression is only used when defining a generator function, and can only be used in the body of a function definition. Using a *yield* expression in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of a generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first *yield* expression, where it is suspended again, returning the value of *expression_list* to generator's caller. By suspended we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the *yield* expression was just another external call. The value of the

yield expression after resuming depends on the method which resumed the execution.

All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where should the execution continue after it yields; the control is always transferred to the generator's caller.

제너레이터-이터레이터 메서드

이 서브섹션은 제너레이터 이터레이터의 메서드들을 설명한다. 제너레이터 함수의 실행을 제어하는데 사용될 수 있다.

제너레이터가 이미 실행 중일 때 아래에 나오는 메서드들을 호출하면 `ValueError` 예외를 일으키는 것에 주의해야 한다.

`generator.`**`next`**`()`
: Starts the execution of a generator function or resumes it at the last executed *yield* expression. When a generator function is resumed with a *next()* method, the current *yield* expression always evaluates to `None`. The execution then continues to the next *yield* expression, where the generator is suspended again, and the value of the *expression_list* is returned to *next()*'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

`generator.`**`send`**`(`*value*`)`
: Resumes the execution and 《sends》 a value into the generator function. The `value` argument becomes the result of the current *yield* expression. The *send()* method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value. When *send()* is called to start the generator, it must be called with `None` as the argument, because there is no *yield* expression that could receive the value.

`generator.`**`throw`**`(`*type*`[,` *value*`[,` *traceback*`]])`
: Raises an exception of type `type` at the point where generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

`generator.`**`close`**`()`
: Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function then raises `StopIteration` (by exiting normally, or due to already being closed) or `GeneratorExit` (by not catching the exception), close returns to its caller. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. *close()* does nothing if the generator has already exited due to an exception or normal exit.

여기에 제너레이터와 제너레이터 함수의 동작을 시연하는 간단한 예가 있다:

```
>>> def echo(value=None):
...     print "Execution starts when 'next()' is called for the first time."
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception, e:
...                 value = e
...     finally:
...         print "Don't forget to clean up when 'close()' is called."
...
>>> generator = echo(1)
>>> print generator.next()
Execution starts when 'next()' is called for the first time.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
1
>>> print generator.next()
None
>>> print generator.send(2)
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

더 보기:

PEP 342 - 개선된 제너레이터를 통한 코루틴 제너레이터의 API와 문법을 개선해서, 간단한 코루틴으로 사용할 수 있도록 만드는 제안.

5.3 프라이머리

프라이머리는 언어에서 가장 강하게 결합하는 연산들을 나타낸다. 문법은 이렇다:

```
primary ::= atom | attributeref | subscription | slicing | call
```

5.3.1 어트리뷰트 참조

어트리뷰트 참조는 마침표(period)와 이름이 뒤에 붙은 프라이머리다:

```
attributeref ::= primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, e.g., a module, list, or an instance. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

5.3.2 서브스크립션(Subscriptions)

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

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

The primary must evaluate to an object of a sequence or mapping type.

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

If the primary is a sequence, the expression list must evaluate to a plain integer. If this value is negative, the length of the sequence is added to it (so that, e.g., `x[-1]` selects the last item of `x`.) The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero).

문자열의 항목은 문자다. 문자는 별도의 데이터형이 아니고, 하나의 문자만을 가진 문자열이다.

5.3.3 슬라이싱(Slicings)

슬라이싱은 시퀀스 객체 (예를 들어, 문자열 튜플 리스트)에서 어떤 범위의 항목들을 선택한다. 슬라이싱은 표현식이나 대입의 타깃이나 *del* 문에 사용될 수 있다. 슬라이싱의 문법은 이렇다:

```
slicing            ::=  simple_slicing | extended_slicing
simple_slicing     ::=  primary "[" short_slice "]"
extended_slicing   ::=  primary "[" slice_list "]"
slice_list         ::=  slice_item ("," slice_item)* [","]
slice_item         ::=  expression | proper_slice | ellipsis
proper_slice       ::=  short_slice | long_slice
short_slice        ::=  [lower_bound] ":" [upper_bound]
long_slice         ::=  short_slice ":" [stride]
lower_bound        ::=  expression
upper_bound        ::=  expression
stride             ::=  expression
ellipsis           ::=  "..."
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice nor ellipses). Similarly, when the slice list has exactly one short slice and no trailing comma, the interpretation as a simple slicing takes priority over that as an extended slicing.

The semantics for a simple slicing are as follows. The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers; defaults are zero and the `sys.maxint`, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index *k* such that `i <= k < j` where *i* and *j* are the specified lower and upper bounds. This may be an empty sequence. It is not an error if *i* or *j* lie outside the range of valid indexes (such items don't exist so they aren't selected).

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in `Ellipsis` object. The conversion of a proper slice is a slice object (see section 표준형 계층) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

5.3.4 호출

호출은 콜러블 객체 (예를 들어, 함수) 를 빌 수도 있는 인자 들의 목록으로 호출한다.

```
call                 ::=  primary "(" [argument_list [","]
                          | expression genexpr_for] ")"
argument_list        ::=  positional_arguments ["," keyword_arguments]
                          ["," "*" expression] ["," keyword_arguments]
                          ["," "**" expression]
                          | keyword_arguments ["," "*" expression]
                          ["," "**" expression]
```

```
                                | "*" expression ["," keyword_arguments] ["," "**" expression]
                                | "**" expression
positional_arguments    ::=     expression ("," expression)*
keyword_arguments       ::=     keyword_item ("," keyword_item)*
keyword_item            ::=     identifier "=" expression
```

A trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and certain class instances themselves are callable; extensions may define additional callable object types). All argument expressions are evaluated before the call is attempted. Please refer to section 함수 정의 for the syntax of formal *parameter* lists.

키워드 인자가 있으면, 먼저 다음과 같이 위치 인자로 변환된다. 먼저 형식 파라미터들의 채워지지 않은 슬롯들의 목록이 만들어진다. N 개의 위치 인자들이 있다면, 처음 N 개의 슬롯에 넣는다. 그다음, 각 키워드 인자마다, 식별자가 대응하는 슬롯을 결정하는 데 사용된다 (식별자가 첫 번째 형식 파라미터의 이름과 같으면, 첫 번째 슬롯은 사용되고, 이런 식으로 계속한다). 슬롯이 이미 채워졌으면, `TypeError` 예외를 일으킨다. 그렇지 않으면 그 인자의 값을 슬롯에 채워 넣는다 (표현식이 `None` 이라 할지라도, 슬롯을 채우게 된다). 모든 인자가 처리되었을 때, 아직 채워지지 않은 슬롯들을 함수 정의로부터 오는 대응하는 기본값들로 채운다. (기본값들은 함수가 정의될 때 한 번만 값을 구한다; 그래서, 리스트나 딕셔너리 같은 가변객체들이 기본값으로 사용되면 해당 슬롯에 인자값을 지정하지 않은 모든 호출에서 공유된다; 보통 이런 상황은 피해야 할 일이다.) 만약 기본값이 지정되지 않고, 아직도 비어있는 슬롯이 남아있다면, `TypeError` 예외가 발생한다. 그렇지 않으면, 채워진 슬롯의 목록이 호출의 인자 목록으로 사용된다.

구현은 위치 파라미터가 이름을 갖지 않아서, 설사 문서화의 목적으로 이름이 붙여졌다 하더라도, 키워드로 공급될 수 없는 내장 함수들을 제공할 수 있다. CPython 에서, 인자들을 파싱하기 위해 `PyArg_ParseTuple()` 를 사용하는 C로 구현된 함수들이 이 경우다.

형식 파라미터 슬롯들보다 많은 위치 인자들이 있으면, `*identifier` 문법을 사용하는 형식 파라미터가 있지 않은 한, `TypeError` 예외를 일으킨다; 이 경우, 그 형식 파라미터는 남는 위치 인자들을 포함하는 튜플을 전달받는다 (또는 남는 위치 인자들이 없으면 빈 튜플).

키워드 인자가 형식 파라미터 이름에 대응하지 않으면, `**identifier` 문법을 사용하는 형식 파라미터가 있지 않은 한, `TypeError` 예외를 일으킨다; 이 경우, 그 형식 파라미터는 남는 키워드 인자들을 포함하는 딕셔너리나, 남는 위치기반 인자들이 없으면 빈 (새) 딕셔너리를 전달받는다.

If the syntax `*expression` appears in the function call, `expression` must evaluate to an iterable. Elements from this iterable are treated as if they were additional positional arguments; if there are positional arguments *x1*, ..., *xN*, and `expression` evaluates to a sequence *y1*, ..., *yM*, this is equivalent to a call with M+N positional arguments *x1*, ..., *xN*, *y1*, ..., *yM*.

A consequence of this is that although the `*expression` syntax may appear *after* some keyword arguments, it is processed *before* the keyword arguments (and the `**expression` argument, if any – see below). So:

```
>>> 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 않는다.

If the syntax `**expression` appears in the function call, `expression` must evaluate to a mapping, the contents of which are treated as additional keyword arguments. In the case of a keyword appearing in both `expression` and as an explicit keyword argument, a `TypeError` exception is raised.

Formal parameters using the syntax `*identifier` or `**identifier` cannot be used as positional argument slots or as keyword argument names. Formal parameters using the syntax `(sublist)` cannot be used as keyword argument names; the outermost sublist corresponds to a single unnamed argument slot, and the argument value is assigned to the sublist using the usual tuple assignment rules after all other parameter processing is done.

호출은 예외를 일으키지 않는 한, 항상 어떤 값을 돌려준다, `None` 일 수 있다. 이 값이 어떻게 계산되는지는 콜러블 객체의 형에 달려있다.

만약 그것이—

사용자 정의 함수면: 인자 목록을 전달해서 함수의 코드 블록이 실행된다. 코드 블록이 처음으로 하는 일은 형식 파라미터들을 인자에 결합하는 것이다; 이것은 섹션 함수 정의 에서 설명한다. 코드 블록이 *return* 문을 실행하면, 함수 호출의 반환 값을 지정하게 된다.

내장 함수나 메서드면: 결과는 인터프리터에 달려있다; 내장 함수와 메서드들에 대한 설명은 built-in-funcs 를 보면 된다.

클래스 객체면: 그 클래스의 새 인스턴스가 반환된다.

클래스 인스턴스 메서드면: 대응하는 사용자 정의 함수가 호출되는데, 그 인스턴스가 첫 번째 인자가 되는 하나만큼 더 긴 인자 목록이 전달된다.

클래스 인스턴스면: 그 클래스는 *__call__()* 메서드를 정의해야 한다; 그 효과는 그 메서드가 호출되는 것과 같다.

5.4 거듭제곱 연산자

거듭제곱 연산자는 그것의 왼쪽에 붙는 일 항 연산자보다 더 강하게 결합한다; 그것의 오른쪽에 붙는 일 항 연산자보다는 약하게 결합한다. 문법은 이렇다:

```
power  ::=  primary ["**" u_expr]
```

그래서, 괄호가 없는 거듭제곱과 일 항 연산자의 시퀀스에서, 연산자는 오른쪽에서 왼쪽으로 값이 구해진다 (이것이 피연산자의 값을 구하는 순서를 제약하는 것은 아니다): `-1**2` 은 `-1` 이 된다.

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type. The result type is that of the arguments after coercion.

With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised).

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `ValueError`.

5.5 일 항 산술과 비트 연산

모든 일 항 산술과 비트 연산자는 같은 우선순위를 갖는다.

```
u_expr  ::=  power | "-" u_expr | "+" u_expr | "~" u_expr
```

일 항 - (마이너스) 연산자는 그 숫자 인자의 음의 값을 준다.

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

The unary ~ (invert) operator yields the bitwise inversion of its plain or long integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers.

세 가지 경우 모두, 인자가 올바른 형을 갖지 않는다면, `TypeError` 예외가 발생한다.

5.6 이항 산술 연산

이항 산술 연산자는 관습적인 우선순위를 갖는다. 이 연산자 중 일부는 일부 비 숫자 형에도 적용됨에 주의해야 한다. 거듭제곱 연산자와는 별개로, 오직 두 가지 수준만 있는데, 하나는 곱셈형 연산자들이고, 하나는 덧셈형 연산자들이다.

```
m_expr  ::=  u_expr | m_expr "*" u_expr | m_expr "//" u_expr | m_expr "/" u_expr
             | m_expr "%" u_expr
a_expr  ::=  m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer (plain or long) and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the ⟨floor⟩ function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

`%` (모듈로, modulo) 연산자는 첫 번째 인자를 두 번째 인자로 나눈 나머지를 준다. 숫자 인자들은 먼저 공통형으로 변환된다. 오른쪽 인자가 0이면 `ZeroDivisionError` 예외를 일으킨다. 인자들은 실수가 될 수 있다, 예를 들어, `3.14%0.7` 는 `0.34` 와 같다 (`3.14` 가 `4*0.7 + 0.34` 와 같으므로.) 모듈로 연산자는 항상 두 번째 피연산자와 같은 부호를 갖는 결과를 준다 (또는 0이다); 결과의 절댓값은 두 번째 피연산자의 절댓값보다 작다[2].

The integer division and modulo operators are connected by the following identity: `x == (x/y)*y + (x%y)`. Integer division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x/y, x%y)`. These identities don't hold for floating point numbers; there similar identities hold approximately where `x/y` is replaced by `floor(x/y)` or `floor(x/y) - 1`[3].

In addition to performing the modulo operation on numbers, the `%` operator is also overloaded by string and unicode

[2] `abs(x%y) < abs(y)` 이 수학적으로는 참이지만, float의 경우에는 소수점 자름 (roundoff) 때문에 수치적으로 참이 아닐 수 있다. 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 값이다. 함수 `math.fmod()` 는 부호가 첫 번째 인자의 부호에 맞춰진 결과를 주기 때문에, 이 경우 `-1e-100` 을 돌려준다. 어떤 접근법이 더 적절한지는 응용 프로그램에 달려있다.

[3] If x is very close to an exact integer multiple of y, it's possible for `floor(x/y)` to be one larger than `(x-x%y)/y` due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

objects to perform string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section string-formatting.

버전 2.3부터 폐지: The floor division operator, the modulo operator, and the `divmod()` function are no longer defined for complex numbers. Instead, convert to a floating point number using the `abs()` function if appropriate.

The + (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

– (빼기) 연산자는 그 인자들의 차를 준다. 숫자 인자들은 먼저 공통형으로 변환된다.

5.7 시프트 연산

시프트 연산은 산술 연산보다 낮은 우선순위를 갖는다.

```
shift_expr  ::=   a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

These operators accept plain or long integers as arguments. The arguments are converted to a common type. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by *n* bits is defined as division by `pow(2, n)`. A left shift by *n* bits is defined as multiplication with `pow(2, n)`. Negative shift counts raise a `ValueError` exception.

참고: 현재 구현에서, 우측 피연산자는 최대 `sys.maxsize` 일 것이 요구된다. 우측 피연산자가 `sys.maxsize` 보다 크면 `OverflowError` 예외가 발생한다.

5.8 이항 비트 연산

세 개의 비트 연산은 각기 다른 우선순위를 갖는다:

```
and_expr  ::=   shift_expr | and_expr "&" shift_expr
xor_expr  ::=   and_expr | xor_expr "^" and_expr
or_expr   ::=   xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be plain or long integers. The arguments are converted to a common type.

5.9 비교

C와는 달리, 파이썬에서 모든 비교 연산은 같은 우선순위를 갖는데, 산술, 시프팅, 비트 연산들보다 낮다. 또한, C와는 달리, `a < b < c` 와 같은 표현식이 수학에서와 같은 방식으로 해석된다.

```
comparison    ::=  or_expr ( comp_operator or_expr )*
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
                   | "is" ["not"] | ["not"] "in"
```

비교는 논리값을 준다: `True` 또는 `False`

비교는 자유롭게 연결될 수 있다, 예를 들어, `x < y <= z` 는 `x < y and y <= z` 와 동등한데, 차이점은 `y` 의 값을 오직 한 번만 구한다는 것이다 (하지만 두 경우 모두 `x < y` 가 거짓이면 `z` 의 값은 구하지 않는다).

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 동등하다.

`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 한다.

The forms <> and != are equivalent; for consistency with C, != is preferred; where != is mentioned below <> is also accepted. The <> spelling is considered obsolescent.

5.9.1 값 비교

연산자 <, >, ==, >=, <=, != 는 두 객체의 값을 비교한다. 객체들이 같은 형일 필요는 없다.

객체, 값, 형 장은 객체들이 (형과 아이덴티티에 더해) 값을 갖는다고 말하고 있다. 파이썬에서 객체의 값은 좀 추상적인 개념이다: 예를 들어, 객체의 값에 대한 규범적인 (canonical) 액세스 방법은 없다. 또한, 객체의 값이 특별한 방식 (예를 들어, 모든 데이터 어트리뷰트로 구성되는 것) 으로 구성되어야 한다는 요구 사항도 없다. 비교 연산자는 객체의 값이 무엇인지에 대한 특정한 종류의 개념을 구현한다. 객체의 값을 비교를 통해 간접적으로 정의한다고 생각해도 좋다.

Types can customize their comparison behavior by implementing a *__cmp__()* method or *rich comparison methods* like *__lt__()*, described in 기본적인 커스터마이제이션.

동등 비교 (== 와 !=) 의 기본 동작은 객체의 아이덴티티에 기반을 둔다. 그래서, 같은 아이덴티티를 갖는 인스턴스 간의 동등 비교는 같음을 주고, 다른 아이덴티티를 갖는 인스턴스 간의 동등 비교는 다름을 준다. 이 기본 동작의 동기는 모든 객체가 반사적 (reflexive) (즉, `x is y` 는 `x == y` 를 암시한다) 이도록 만들고자 하는 욕구다.

The default order comparison (<, >, <=, and >=) gives a consistent but arbitrary order.

(This unusual definition of comparison was used to simplify the definition of operations like sorting and the *in* and *not in* operators. In the future, the comparison rules for objects of different types are likely to change.)

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 있다. 그런 형들은 자신의 비교 동작을 커스터마이즈 할 필요가 있고, 사실 많은 내장형이 그렇게 하고 있다.

다음 목록은 가장 중요한 내장형들의 비교 동작을 기술한다.

- 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 가능하다. 관련된 형들의 한계 안에서, 정밀도의 손실 없이 수학적으로 (알고리즘 적으로) 올바르게 비교한다.

- Strings (instances of `str` or `unicode`) compare lexicographically using the numeric equivalents (the result of the built-in function `ord()`) of their characters.[4] When comparing an 8-bit string and a Unicode string, the 8-bit string is converted to Unicode. If the conversion fails, the strings are considered unequal.
- Instances of `tuple` or `list` can be compared only within each of their types. Equality comparison across these types results in unequality, and ordering comparison across these types gives an arbitrary order.

  These sequences compare lexicographically using comparison of corresponding elements, whereby reflexivity of the elements is enforced.

  In enforcing reflexivity of elements, the comparison of collections assumes that for a collection element `x`, `x == x` is always true. Based on that assumption, element identity is compared first, and element comparison is performed only for distinct elements. This approach yields the same result as a strict element comparison would, if the compared elements are reflexive. For non-reflexive elements, the result is different than for strict element comparison.

  내장 컬렉션들의 사전적인 비교는 다음과 같이 이루어진다:

  - 두 컬렉션이 같다고 비교되기 위해서는, 같은 형이고, 길이가 같고, 대응하는 요소들의 각 쌍이 같다고 비교되어야 한다 (예를 들어, `[1,2] == (1,2)` 는 거짓인데, 형이 다르기 때문이다).
  - Collections are ordered the same as their first unequal elements (for example, `cmp([1,2,x], [1,2,y])` returns the same as `cmp(x,y)`). If a corresponding element does not exist, the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).
- 매핑들 (`dict` 의 인스턴스들) 은 같은 *(key, value)* 쌍들을 가질 때, 그리고 오직 이 경우만 같다고 비교된다. 키와 값의 동등 비교는 반사성을 강제한다.

  Outcomes other than equality are resolved consistently, but are not otherwise defined.[5]
- Most other objects of built-in types compare unequal unless they are the same object; the choice whether one object is considered smaller or larger than another one is made arbitrarily but consistently within one execution of a program.

비교 동작을 커스터마이즈하는 사용자 정의 클래스들은 가능하다면 몇 가지 일관성 규칙을 준수해야 한다:

- 동등 비교는 반사적 (reflexive) 이어야 한다. 다른 말로 표현하면, 아이덴티티가 같은 객체는 같다고 비교되어야 한다:

  `x is y` 면 `x == y` 다.
- 비교는 대칭적 (symmetric) 이어야 한다. 다른 말로 표현하면, 다음과 같은 표현식은 같은 결과를 주어야 한다:

  `x == y` 와 `y == x`

  `x != y` 와 `y != x`

  `x < y` 와 `y > x`

  `x <= y` 와 `y >= x`
- 비교는 추이적 (transitive) 이어야 한다. 다음 (철저하지 않은) 예들이 이것을 예증한다:

[4] 유니코드 표준은 코드 포인트 (*code points*) (예를 들어, U+0041) 와 추상 문자 (*abstract characters*) (예를 들어, 《LATIN CAPITAL LETTER A》) 를 구분한다. 유니코드에 있는 대부분의 추상 문자들이 오직 하나의 코드 포인트만으로 표현되지만, 추가로 하나 이상의 코드 포인트의 시퀀스로 표현될 수 있는 추상 문자들이 많이 있다. 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 있다.

The comparison operators on unicode strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, `u"\u00C7" == u"\u0043\u0327"` is `False`, even though both strings represent the same abstract character 《LATIN CAPITAL LETTER C WITH CEDILLA》.

문자열을 추상 문자 수준에서 비교하려면 (즉, 사람에게 직관적인 방법으로), `unicodedata.normalize()` 를 사용하라.

[5] Earlier versions of Python used lexicographic comparison of the sorted (key, value) lists, but this was very expensive for the common case of comparing for equality. An even earlier version of Python compared dictionaries by identity only, but this caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to `{}`.

`x > y and y > z` 면 `x > z` 다

x < y and y <= z“ 면 `x < z` 다

- 역 비교는 논리적 부정이 되어야 한다. 다른 말로 표현하면, 다음 표현식들이 같은 값을 주어야 한다:

  `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 적용된다 (예를 들어, 시퀀스에는 적용되지만, 집합과 매핑은 그렇지 않다). `total_ordering()` 데코레이터 또한 보기 바란다.

- `hash()` 결과는 동등성과 일관성을 유지해야 한다. 같은 객체들은 같은 해시값을 갖거나 해시 불가능으로 지정되어야 한다.

Python does not enforce these consistency rules.

5.9.2 멤버십 검사 연산

연산자 *in* 과 *not in* 은 멤버십을 검사한다. `x in s` 는 *x* 가 *s* 의 멤버일 때 `True` 를, 그렇지 않을 때 `False` 를 준다. `x not in s` 은 `x in s` 의 부정을 준다. 딕셔너리 뿐만 아니라 모든 내장 시퀀스들과 집합 형들이 이것을 지원하는데, 딕셔너리의 경우는 *in* 이 딕셔너리에 주어진 키가 있는지 검사한다. 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 동등하다.

문자열과 바이트열 형의 경우, `x in y` 는 *x* 가 *y* 의 서브 스트링 (substring) 인 경우, 그리고 오직 그 경우만 `True` 다. 동등한 검사는 `y.find(x) != -1` 다. 빈 문자열은 항상 다른 문자열들의 서브 스트링으로 취급되기 때문에, `"" in "abc"` 은 `True` 를 돌려준다.

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

__contains__() 를 정의하지 않지만 *__iter__()* 를 정의하는 사용자 정의 클래스의 경우, `x in y` 는 `y` 를 탐색할 때 `x == z` 를 만족하는 어떤 값 `z` 가 만들어지면 `True` 다. 탐색하는 동안 예외가 발생하면 *in* 이 그 예외를 일으킨 것으로 취급된다.

Lastly, the old-style iteration protocol is tried: if a class defines *__getitem__()*, `x in y` is `True` if and only if there is a non-negative integer index *i* such that `x == y[i]`, and all lower integer indices do not raise `IndexError` exception. (If any other exception is raised, it is as if *in* raised that exception).

연산자 *not in* 은 *in* 의 논리적 부정으로 정의된다.

5.9.3 아이덴티티 비교

The operators *is* and *is not* test for object identity: `x is y` is true if and only if *x* and *y* are the same object. `x is not y` yields the inverse truth value.[6]

[6] 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 있다. 더 자세한 정보는 그들의 문서를 확인하기 바란다.

5.10 논리 연산(Boolean operations)

```
or_test   ::=   and_test | or_test "or" and_test
and_test  ::=   not_test | and_test "and" not_test
not_test  ::=   comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. (See the *__nonzero__()* special method for a way to change this.)

연산자 *not* 은 그 인자가 거짓이면 `True` 를, 그렇지 않으면 `False` 를 준다.

표현식 `x and y` 는 먼저 *x* 의 값을 구한다; *x* 가 거짓이면 그 값을 돌려준다; 그렇지 않으면 *y* 의 값을 구한 후에 그 결과를 돌려준다.

표현식 `x or y` 는 먼저 *x* 의 값을 구한다; *x* 가 참이면 그 값을 돌려준다. 그렇지 않으면 *y* 의 값을 구한 후에 그 결과를 돌려준다.

(Note that neither *and* nor *or* restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because *not* has to invent a value anyway, it does not bother to return a value of the same type as its argument, so e.g., `not 'foo'` yields `False`, not `''`.)

5.11 Conditional Expressions

버전 2.5에 추가.

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

조건 표현식은 (때로 《삼 항 연산자(ternary operator)》라고 불린다) 모든 파이썬 연산에서 가장 낮은 우선순위를 갖는다.

The expression `x if C else y` first evaluates the condition, *C* (*not x*); if *C* is true, *x* is evaluated and its value is returned; otherwise, *y* is evaluated and its value is returned.

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

5.12 람다(Lambdas)

```
lambda_expr      ::=   "lambda" [parameter_list]: expression
old_lambda_expr  ::=   "lambda" [parameter_list]: old_expression
```

Lambda expressions (sometimes called lambda forms) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `lambda parameters: expression` yields a function object. The unnamed object behaves like a function object defined with

```
def <lambda>(parameters):
    return expression
```

See section 함수 정의 for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements.

5.13 표현식 목록(Expression lists)

```
expression_list  ::=  expression ( "," expression )* [","]
```

An expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

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

5.14 값을 구하는 순서

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

다 줄들에서, 표현식은 그들의 끝에 붙은 숫자들의 순서대로 값이 구해진다:

```
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
```

5.15 연산자 우선순위

The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right — see section 비교 — and exponentiation, which groups from right to left).

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 포함한다
|	비트 OR
^	비트 XOR
&	비트 AND
<<, >>	시프트
+, –	덧셈과 뺄셈
*, /, //, %	Multiplication, division, remainder[7]
+x, –x, ~x	양, 음, 비트 NOT
**	거듭제곱[8]
x[index], x[index:index], x(arguments...), x.attribute	서브스크립션, 슬라이싱, 호출, 어트리뷰트 참조
(expressions...), [expressions...], {key: value...}, `expressions...`	Binding or tuple display, list display, dictionary display, string conversion

[7] % 연산자는 문자열 포매팅에도 사용된다; 같은 우선순위가 적용된다.
[8] 거듭제곱 연산자 ** 는 오른쪽에 오는 산술이나 비트 일 항 연산자보다 약하게 결합한다, 즉, 2**-1 는 0.5 다.

CHAPTER 6

단순문 (Simple statements)

Simple statements are comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt  ::=  expression_stmt
                  | assert_stmt
                  | assignment_stmt
                  | augmented_assignment_stmt
                  | pass_stmt
                  | del_stmt
                  | print_stmt
                  | return_stmt
                  | yield_stmt
                  | raise_stmt
                  | break_stmt
                  | continue_stmt
                  | import_stmt
                  | future_stmt
                  | global_stmt
                  | exec_stmt
```

6.1 표현식 문

표현식 문은 값을 계산하고 출력하거나, (보통) 프로시저 (procedure) (의미 없는 결과를 돌려주는 함수; 파이썬에서 프로시저는 `None` 값을 돌려준다)를 호출하기 위해 (대부분 대화형으로) 사용된다. 표현식 문의 다른 사용도 허락되고 때때로 쓸모가 있다.

```
expression_stmt  ::=  expression_list
```

표현식 문은 (하나의 표현식일 수 있는) 표현식 목록의 값을 구한다.

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output (see section *The print statement*) on a line by itself. (Expression statements yielding `None` are not written, so that procedure calls do not cause any output.)

6.2 대입문

대입문은 이름을 값에 (재) 연결하고 가변 객체의 어트리뷰트나 항목들을 수정한다.

```
assignment_stmt  ::=  (target_list "=")+ (expression_list | yield_expression)
target_list      ::=  target ("," target)* [","]
target           ::=  identifier
                      | "(" target_list ")"
                      | "[" [target_list] "]"
                      | attributeref
                      | subscription
                      | slicing
```

(See section 프라이머리 for the syntax definitions for the last three symbols.)

대입문은 표현식 목록 (이것이 하나의 표현식일 수도, 쉼표로 분리된 목록일 수도 있는데, 후자의 경우는 튜플이 만들어진다는 것을 기억하라) 의 값을 구하고, 왼쪽에서 오른쪽으로, 하나의 결과 객체를 타깃 목록의 각각에 대입한다.

대입은 타깃 (목록) 의 형태에 따라 재귀적으로 정의된다. 타깃이 가변 객체의 일부 (어트리뷰트 참조나 서브스크립션이나 슬라이싱) 면, 가변 객체가 최종적으로 대입을 수행해야만 하고, 그것이 올바른지 아닌지를 결정하고, 대입이 받아들여질 수 없으면 예외를 일으킬 수 있다. 다양한 형들이 주시하는 규칙들과 발생하는 예외들은 그 객체 형의 정의에서 주어진다 (표준형 계층 섹션을 보라).

Assignment of an object to a target list is recursively defined as follows.

- If the target list is a single target: The object is assigned to that target.
- If the target list is a comma-separated list of targets: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

하나의 타깃에 대한 객체의 대입은 다음과 같이 재귀적으로 정의된다.

- 타깃이 식별자 (이름) 면:
  - If the name does not occur in a *global* statement in the current code block: the name is bound to the object in the current local namespace.
  - Otherwise: the name is bound to the object in the current global namespace.

  그 이름이 이미 연결되어 있으면 재연결된다. 이것은 기존에 연결되어 있던 객체의 참조 횟수가 0 이 되도록 만들어서, 객체가 점유하던 메모리가 반납되고 파괴자 (destructor) (갖고 있다면) 가 호출되도록 만들 수 있다.
- If the target is a target list enclosed in parentheses or in square brackets: The object must be an iterable with the same number of items as there are targets in the target list, and its items are assigned, from left to right, to the corresponding targets.
- 타깃이 어트리뷰트 참조면: 참조의 프라이머리 표현식의 값을 구한다. 이것은 대입 가능한 어트리뷰트를 가진 객체를 주어야 하는데, 그렇지 않으면 `TypeError` 가 일어난다. 그에 그 객체에 주어진 어트리뷰

트로 객체를 대입하도록 요청한다; 대입을 수행할 수 없다면 예외 (보통 `AttributeError` 이지만, 꼭 그럴 필요는 없다) 를 일으킨다.

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 있다. LHS 타깃 `a.x` 는 항상 필요하면 만들어서라도 항상 인스턴스 어트리뷰트를 설정한다. 그래서, 두 `a.x` 가 같은 어트리뷰트를 가리키는 것은 필요조건이 아니다: RHS 표현식이 클래스 어트리뷰트를 가리킨다면, LHS 는 대입의 타깃으로 새 인스턴스 어트리뷰트를 만든다:

```
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 없다.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (such as a list), the subscript must yield a plain integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

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

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to (small) integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

현재 구현에서, 타깃의 문법은 표현식과 같게 유지되고, 잘못된 문법은 코드 생성 단계에서 거부되기 때문에 에러 메시지가 덜 상세해지는 결과를 낳고 있다.

WARNING: Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 〈safe〉 (for example `a, b = b, a` swaps two variables), overlaps *within* the collection of assigned-to variables are not safe! For instance, the following program prints `[0, 2]`:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

6.2.1 증분 대입문(Augmented assignment statements)

증분 대입문은 한 문장에서 이항 연산과 대입문을 합치는 것이다:

```
augmented_assignment_stmt   ::=   augtarget augop (expression_list | yield_expression)
augtarget                   ::=   identifier | attributeref | subscription | slicing
augop                       ::=   "+=" | "-=" | "*=" | "/=" | "//=" | "%=" | "**="
                                  | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section 프라이머리 for the syntax definitions for the last three symbols.)

증분 대입은 타깃(일반 대입문과는 달리 언 패킹이 될 수 없다)과 표현식 목록의 값을 구하고, 둘을 피연산자로 삼아 대입의 형에 맞는 이항 연산을 수행한 후, 원래의 타깃에 그 결과를 대입한다. 타깃은 오직 한 번만 값이 구해진다.

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

하나의 문장에서 튜플과 다중 타깃으로 대입하는 것을 예외로 하면, 증분 대입문에 의한 대입은 일반 대입과 같은 방법으로 처리된다. 마찬가지로, 제자리 동작의 가능성을 예외로 하면, 증분 대입 때문에 수행되는 이진 연산은 일반 이진 연산과 같다.

어트리뷰트 참조인 타깃의 경우, 일반 대입처럼 클래스와 인스턴스 어트리뷰트에 관한 경고 가 적용된다.

6.3 `assert` 문

assert 문은 프로그램에 디버깅 어서션(debugging assertion)을 삽입하는 편리한 방법이다:

```
assert_stmt   ::=   "assert" expression ["," expression]
```

간단한 형태, `assert expression` 은 다음과 동등하다

```
if __debug__:
    if not expression: raise AssertionError
```

확장된 형태, `assert expression1, expression2` 는 다음과 동등하다

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

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

`__debug__` 에 대한 대입은 허락되지 않는다. 이 내장 변수의 값은 인터프리터가 시작할 때 결정된다.

6.4 pass 문

```
pass_stmt   ::=   "pass"
```

pass 는 널(null) 연산이다 — 실행될 때, 아무런 일도 일어나지 않는다. 문법적으로 문장이 필요하기는 하지만 할 일은 없을 때, 자리를 채우는 용도로 쓸모가 있다, 예를 들어:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

6.5 del 문

```
del_stmt   ::=   "del" target_list
```

삭제는 대입이 정의된 방식과 아주 비슷하게 재귀적으로 정의된다. 전체 세부 사항들을 나열하는 대신, 여기 몇 가지 힌트가 있다.

타깃 목록의 삭제는 각 타깃을 왼쪽에서 오른쪽으로 재귀적으로 삭제한다.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a *global* statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

It is illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

어트리뷰트 참조, 서브스크립션, 슬라이싱의 삭제는 관련된 프라이머리 객체로 전달된다; 슬라이싱의 삭제는 일반적으로 우변 형의 빈 슬라이스를 대입하는 것과 동등하다 (하지만 이것조차 슬라이싱 되는 객체가 판단한다).

6.6 The print statement

```
print_stmt   ::=   "print" ([expression ("," expression)* [","]]
                   | ">>" expression [("," expression)+ [","]])
```

print evaluates each expression in turn and writes the resulting object to standard output (see below). If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. A space is written before each object is (converted and) written, unless the output system believes it is positioned at the beginning of a line. This is the case (1) when no characters have yet been written to standard output, (2) when the last character written to standard output is a whitespace character except `' '`, or (3) when the last write operation on standard output was not a *print* statement. (In some cases it may be functional to write an empty string to standard output for this reason.)

참고: Objects which act like file objects but which are not the built-in file objects often do not properly emulate this aspect of the file object's behavior, so it is best not to rely on this.

A `'\n'` character is written at the end, unless the *print* statement ends with a comma. This is the only action if the statement contains just the keyword *print*.

Standard output is defined as the file object named `stdout` in the built-in module `sys`. If no such object exists, or if it

does not have a `write()` method, a `RuntimeError` exception is raised.

print also has an extended form, defined by the second portion of the syntax described above. This form is sometimes referred to as 《*print* chevron.》 In this form, the first expression after the >> must evaluate to a 《file-like》 object, specifically an object that has a `write()` method as described above. With this extended form, the subsequent expressions are printed to this file object. If the first expression evaluates to `None`, then `sys.stdout` is used as the file for output.

6.7 return 문

```
return_stmt  ::=  "return" [expression_list]
```

return 은 문법적으로 클래스 정의에 중첩된 경우가 아니라, 함수 정의에만 중첩되어 나타날 수 있다.

표현식 목록이 있으면 값을 구하고, 그렇지 않으면 `None` 으로 치환된다.

return 은 표현식 목록 (또는 `None`) 을 반환 값으로 해서, 현재의 함수 호출을 떠난다.

return 이 *finally* 절을 가진 *try* 문에서 제어가 벗어나도록 만드는 경우, 함수로부터 진짜로 벗어나기 전에 그 *finally* 절이 실행된다.

In a generator function, the *return* statement is not allowed to include an *expression_list*. In that context, a bare *return* indicates that the generator is done and will cause `StopIteration` to be raised.

6.8 yield 문

```
yield_stmt  ::=  yield_expression
```

The *yield* statement is only used when defining a generator function, and is only used in the body of the generator function. Using a *yield* statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator. The body of the generator function is executed by calling the generator's *next()* method repeatedly until it raises an exception.

When a *yield* statement is executed, the state of the generator is frozen and the value of *expression_list* is returned to *next()*'s caller. By 《frozen》 we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time *next()* is invoked, the function can proceed exactly as if the *yield* statement were just another external call.

As of Python version 2.5, the *yield* statement is now allowed in the *try* clause of a *try* ··· *finally* construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's `close()` method will be called, allowing any pending *finally* clauses to execute.

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

참고: In Python 2.2, the *yield* statement was only allowed when the `generators` feature has been enabled. This `__future__` import statement was used to enable the feature:

```
from __future__ import generators
```

더 보기:

PEP 255 - Simple Generators The proposal for adding generators and the *yield* statement to Python.

PEP 342 - Coroutines via Enhanced Generators The proposal that, among other generator enhancements, proposed allowing *yield* to appear inside a *try* ⋯ *finally* block.

6.9 raise 문

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

If no expressions are present, *raise* re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a TypeError exception is raised indicating that this is an error (if running under IDLE, a Queue.Empty exception is raised instead).

Otherwise, *raise* evaluates the expressions to get three objects, using None as the value of omitted expressions. The first two objects are used to determine the *type* and *value* of the exception.

If the first object is an instance, the type of the exception is the class of the instance, the instance itself is the value, and the second object must be None.

If the first object is a class, it becomes the type of the exception. The second object is used to determine the exception value: If it is an instance of the class, the instance becomes the exception value. If the second object is a tuple, it is used as the argument list for the class constructor; if it is None, an empty argument list is used, and any other object is treated as a single argument to the constructor. The instance so created by calling the constructor is used as the exception value.

If a third object is present and not None, it must be a traceback object (see section 표준형 계층), and it is substituted instead of the current location as the place where the exception occurred. If the third object is present and not a traceback object or None, a TypeError exception is raised. The three-expression form of *raise* is useful to re-raise an exception transparently in an except clause, but *raise* with no expressions should be preferred if the exception to be re-raised was the most recently active exception in the current scope.

예외에 대한 더 많은 정보를 예외 섹션에서 발견할 수 있고, 예외를 처리하는 것에 대한 정보는 *try* 문 섹션에 있다.

6.10 break 문

```
break_stmt  ::=  "break"
```

break 는 문법적으로 *for* 나 *while* 루프에 중첩되어서만 나타날 수 있다. 하지만 그 루프 안의 함수나 클래스 정의에 중첩되지는 않는다.

가장 가까이서 둘러싸고 있는 루프를 종료하고, 그 루프가 *else* 절을 갖고 있다면 건너뛴다(skip).

for 루프가 *break* 로 종료되면, 루프 제어 타깃은 현재값을 유지한다.

break 가 *finally* 절을 가 *try* 문에서 제어가 벗어나도록 만드는 경우, 루프로부터 진짜로 벗어나기 전에 그 *finally* 절이 실행된다.

6.11 continue 문

```
continue_stmt  ::=  "continue"
```

continue 는 문법적으로 *for* 나 *while* 루프에 중첩되어서만 나타날 수 있다. 하지만 그 루프 안의 함수나 클래스 정의 또는 그 루프 내의 *finally* 에 중첩되지는 않는다. 가장 가까이서 둘러싸고 있는 루프가 다음 사이클로 넘어가도록 만든다.

continue 가 *finally* 절을 가진 *try* 문에서 제어가 벗어나도록 만드는 경우, 다음 루트 사이클을 시작하기 전에 그 *finally* 절이 실행된다.

6.12 임포트(import) 문

```
import_stmt      ::=  "import" module ["as" name] ( "," module ["as" name] )*
                      | "from" relative_module "import" identifier ["as" name]
                      ( "," identifier ["as" name] )*
                      | "from" relative_module "import" "(" identifier ["as" name]
                      ( "," identifier ["as" name] )* [","] ")"
                      | "from" module "import" "*"
module           ::=  (identifier ".")* identifier
relative_module  ::=  "."* module | "."+
name             ::=  identifier
```

Import statements are executed in two steps: (1) find a module, and initialize it if necessary; (2) define a name or names in the local namespace (of the scope where the *import* statement occurs). The statement comes in two forms differing on whether it uses the *from* keyword. The first form (without *from*) repeats these steps for each identifier in the list. The form with *from* performs step (1) once, and then performs step (2) repeatedly.

To understand how step (1) occurs, one must first understand how Python handles hierarchical naming of modules. To help organize modules and provide a hierarchy in naming, Python has a concept of packages. A package can contain other packages and modules while modules cannot contain other modules or packages. From a file system perspective, packages are directories and modules are files.

Once the name of the module is known (unless otherwise specified, the term《module》will refer to both packages and modules), searching for the module or package can begin. The first place checked is `sys.modules`, the cache of all modules that have been imported previously. If the module is found there then it is used in step (2) of import.

If the module is not found in the cache, then `sys.meta_path` is searched (the specification for `sys.meta_path` can be found in **PEP 302**). The object is a list of *finder* objects which are queried in order as to whether they know how to load the module by calling their `find_module()` method with the name of the module. If the module happens to be contained within a package (as denoted by the existence of a dot in the name), then a second argument to `find_module()` is given as the value of the `__path__` attribute from the parent package (everything up to the last dot in the name of the module being imported). If a finder can find the module it returns a *loader* (discussed later) or returns `None`.

If none of the finders on `sys.meta_path` are able to find the module then some implicitly defined finders are queried. Implementations of Python vary in what implicit meta path finders are defined. The one they all do define, though, is one that handles `sys.path_hooks`, `sys.path_importer_cache`, and `sys.path`.

The implicit finder searches for the requested module in the《paths》specified in one of two places (《paths》do not have to be file system paths). If the module being imported is supposed to be contained within a package then the second argument passed to `find_module()`, `__path__` on the parent package, is used as the source of paths. If the module is not contained in a package then `sys.path` is used as the source of paths.

Once the source of paths is chosen it is iterated over to find a finder that can handle that path. The dict at `sys.path_importer_cache` caches finders for paths and is checked for a finder. If the path does not have a finder cached then `sys.path_hooks` is searched by calling each object in the list with a single argument of the path, returning a finder or raises `ImportError`. If a finder is returned then it is cached in `sys.path_importer_cache` and then used for that path entry. If no finder can be found but the path exists then a value of `None` is stored in `sys.path_importer_cache` to signify that an implicit, file-based finder that handles modules stored as individual files should be used for that path. If the path does not exist then a finder which always returns `None` is placed in the cache for the path.

If no finder can find the module then `ImportError` is raised. Otherwise some finder returned a loader whose `load_module()` method is called with the name of the module to load (see **PEP 302** for the original definition of loaders). A loader has several responsibilities to perform on a module it loads. First, if the module already exists in `sys.modules` (a possibility if the loader is called outside of the import machinery) then it is to use that module for initialization and not a new module. But if the module does not exist in `sys.modules` then it is to be added to that dict before initialization begins. If an error occurs during loading of the module and it was added to `sys.modules` it is to be removed from the dict. If an error occurs but the module was already in `sys.modules` it is left in the dict.

The loader must set several attributes on the module. `__name__` is to be set to the name of the module. `__file__` is to be the 《path》 to the file unless the module is built-in (and thus listed in `sys.builtin_module_names`) in which case the attribute is not set. If what is being imported is a package then `__path__` is to be set to a list of paths to be searched when looking for modules and packages contained within the package being imported. `__package__` is optional but should be set to the name of package that contains the module or package (the empty string is used for module not contained in a package). `__loader__` is also optional but should be set to the loader object that is loading the module.

If an error occurs during loading then the loader raises `ImportError` if some other exception is not already being propagated. Otherwise the loader returns the module that was loaded and initialized.

When step (1) finishes without raising an exception, step (2) can begin.

The first form of *import* statement binds the module name in the local namespace to the module object, and then goes on to import the next identifier, if any. If the module name is followed by *as*, the name following *as* is used as the local name for the module.

The *from* form does not bind the module name: it goes through the list of identifiers, looks each one of them up in the module found in step (1), and binds the name in the local namespace to the object thus found. As with the first form of *import*, an alternate local name can be supplied by specifying 《*as* localname》. If a name is not found, `ImportError` is raised. If the list of identifiers is replaced by a star (`'*'`), all public names defined in the module are bound in the local namespace of the *import* statement..

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character (`'_'`). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The *from* form with `*` may only occur in a module scope. If the wild card form of import — `import *` — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError`.

임포트할 모듈을 지정할 때 모듈의 절대 이름(absolute name)을 지정할 필요는 없다. 모듈이나 패키지가 다른 패키지 안에 포함될 때, 같은 상위 패키지 내에서는 그 패키지 이름을 언급할 필요 없이 상대 임포트(relative import)를 할 수 있다. *from* 뒤에 지정되는 패키지나 모듈 앞에 붙이는 점으로, 정확한 이름을 지정하지 않고도 현재 패키지 계층을 얼마나 거슬러 올라가야 하는지 지정할 수 있다. 하나의 점은 이 임포트를 하는 모듈이 존재하는 현재 패키지를 뜻한다. 두 개의 점은 한 패키지 수준을 거슬러 올라가는 것을 뜻한다. 세 개의 점은 두 개의 수준을, 등등이다. 그래서 `pkg` 패키지에 있는 모듈에서 `from . import mod`를 실행하면, `pkg.mod`를 임포트하게 된다. `pkg.subpkg1` 안에서 `from ..subpkg2 import mod`를 실행하면 `pkg.subpkg2.mod`를 임포트하게 된다. 상대 임포트에 대한 규격은 **PEP 328** 안에 들어있다.

`importlib.import_module()` is provided to support applications that determine which modules need to be loaded dynamically.

6.12.1 퓨처 문

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_statement  ::=  "from" "__future__" "import" feature ["as" name]
                       ("," feature ["as" name])*
                       | "from" "__future__" "import" "(" feature ["as" name]
                       ("," feature ["as" name])* [","] ")"
feature           ::=  identifier
name              ::=  identifier
```

퓨처 문은 모듈의 거의 처음에 나와야 한다. 퓨처 문 앞에 나올 수 있는 줄들은:

- 모듈 독스트링 (docstring) (있다면),
- 주석
- 빈 줄, 그리고
- 다른 퓨처 문들

The features recognized by Python 2.6 are `unicode_literals`, `print_function`, `absolute_import`, `division`, `generators`, `nested_scopes` and `with_statement`. `generators`, `with_statement`, `nested_scopes` are redundant in Python version 2.6 and above because they are always enabled.

퓨처 문은 구체적으로는 컴파일 시점에 인식되고 다뤄진다: 핵심 구성물들의 의미에 대한 변경은 종종 다른 코드 생성을 통해 구현된다. 새 기능이 호환되지 않는 (새로운 예약어처럼) 새로운 문법을 도입하는 경우조차 가능한데, 이 경우는 컴파일러가 모듈을 다르게 파싱할 수 있다. 그런 결정들은 실행 시점으로 미뤄질 수 없다..

배포마다, 컴파일러는 어떤 기능 이름들이 정의되어 있는지 알고, 만약 퓨처 문이 알지 못하는 기능을 포함하고 있으면 컴파일 시점 에러를 일으킨다.

직접적인 실행 시점의 개념은 다른 임포트 문들과 같다: 표준 모듈 `__future__`, 후에 설명한다, 다 있고, 퓨처 문이 실행되는 시점에 일반적인 방법으로 임포트된다.

흥미로운 실행 시점의 개념들은 퓨처 문에 의해 활성화되는 구체적인 기능들에 달려있다.

이런 문장에는 아무것도 특별한 것이 없음에 주의해야 한다:

```
import __future__ [as name]
```

이것은 퓨처 문이 아니다; 아무런 특별한 개념이나 문법적인 제약이 없는 평범한 임포트 문일 뿐이다.

Code compiled by an *exec* statement or calls to the built-in functions `compile()` and `execfile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can, starting with Python 2.2 be controlled by optional arguments to `compile()` — see the documentation of that function for details.

대화형 인터프리터 프롬프트에서 입력된 퓨처 문은 인터프리터 세션의 남은 기간 효과를 발생시킨다. 인터프리터가 `-i`, 실행할 스크립트 이름이 전달된다, 옵션으로 시작하고, 그 스크립트가 퓨처 문을 포함하면, 스크립트가 실행된 이후에 시작되는 대화형 세션에서도 효과를 유지한다.

더 보기:

PEP 236 - 백 투 더 **__future__** __future__ 메커니즘에 대한 최초의 제안.

6.13 global 문

```
global_stmt  ::=  "global" identifier ("," identifier)*
```

global 문은 현재 코드 블록 전체에 적용되는 선언이다. 나열된 식별자들이 전역으로 해석되어야 한다는 뜻이다. global 선언 없이 자유 변수들이 전역을 가리킬 수 있기는 하지만, *global* 없이 전역 변수에 값을 대입하는 것은 불가능하다.

global 문에 나열된 이름들은 같은 코드 블록에서 *global* 문 앞에 등장할 수 없다.

Names listed in a *global* statement must not be defined as formal parameters or in a *for* loop control target, *class* definition, function definition, or *import* statement.

CPython implementation detail: The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

Programmer's note: *global* is a directive to the parser. It applies only to code parsed at the same time as the *global* statement. In particular, a *global* statement contained in an *exec* statement does not affect the code block *containing* the *exec* statement, and code contained in an *exec* statement is unaffected by *global* statements in the code containing the *exec* statement. The same applies to the `eval()`, `execfile()` and `compile()` functions.

6.14 The exec statement

```
exec_stmt  ::=  "exec" or_expr ["in" expression ["," expression]]
```

This statement supports dynamic execution of Python code. The first expression should evaluate to either a Unicode string, a *Latin-1* encoded string, an open file object, a code object, or a tuple. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).[1] If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed. For the interpretation of a tuple, see below. In all cases, the code that's executed is expected to be valid as file input (see section 파일 입력). Be aware that the *return* and *yield* statements may not be used outside of function definitions even within the context of code passed to the *exec* statement.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after `in` is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are given as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

The first expression may also be a tuple of length 2 or 3. In this case, the optional parts must be omitted. The form `exec(expr, globals)` is equivalent to `exec expr in globals`, while the form `exec(expr, globals, locals)` is equivalent to `exec expr in globals, locals`. The tuple form of `exec` provides compatibility with Python 3, where `exec` is a function rather than a statement.

버전 2.4에서 변경: Formerly, *locals* was required to be a dictionary.

As a side effect, an implementation may insert additional keys into the dictionaries given besides those corresponding to variable names set by the executed code. For example, the current implementation may add a reference to the dictionary of the built-in module `__builtin__` under the key `__builtins__` (!).

[1] Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use *universal newlines* mode to convert Windows or Mac-style newlines.

Programmer's hints: dynamic evaluation of expressions is supported by the built-in function `eval()`. The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use by *exec*.

CHAPTER 7

복합문 (Compound statements)

복합문은 다른 문장들(의 그룹들)을 포함한다; 어떤 방법으로 그 다른 문장들의 실행에 영향을 주거나 제어한다. 간단하게 표현할 때, 전체 복합문을 한 줄로 쓸 수 있기는 하지만, 일반적으로 복합문은 여러 줄에 걸친다.

The *if*, *while* and *for* statements implement traditional control flow constructs. *try* specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

Compound statements consist of one or more ⟨clauses.⟩ A clause consists of a header and a ⟨suite.⟩ The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which *if* clause a following *else* clause would belong:

```
if test1: if test2: print x
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the *print* statements are executed:

```
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
                    | decorated
suite           ::=   stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=   stmt_list NEWLINE | compound_stmt
stmt_list       ::=   simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a NEWLINE possibly followed by a DEDENT. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the 〈dangling *else*〉 problem is solved in Python by requiring nested *if* statements to be indented).

명확함을 위해 다음에 오는 절들에서 나오는 문법 규칙들은 각 절을 별도의 줄에 놓도록 포매팅한다.

7.1 if 문

if 문은 조건부 실행에 사용된다:

```
if_stmt  ::=  "if" expression ":" suite
              ( "elif" expression ":" suite )*
              ["else" ":" suite]
```

참이 되는 것을 발견할 때까지 표현식들의 값을 하나씩 차례대로 구해서 정확히 하나의 스위트를 선택한다 (참과 거짓의 정의는 논리 연산*(Boolean operations)* 섹션을 보라); 그런 다음 그 스위트를 실행한다 (그리고는 *if* 문의 다른 어떤 부분도 실행되거나 값이 구해지지 않는다). 모든 표현식들이 거짓이면 *else* 절의 스위트가 (있다면) 실행된다.

7.2 while 문

while 문은 표현식이 참인 동안 실행을 반복하는 데 사용된다:

```
while_stmt  ::=  "while" expression ":" suite
                 ["else" ":" suite]
```

이것은 표현식을 반복적으로 검사하고, 참이면, 첫 번째 스위트를 실행한다; 표현식이 거짓이면 (처음부터 거짓일 수도 있다) *else* 절의 스위트가 (있다면) 실행되고 루프를 종료한다.

첫 번째 스위트에서 실행되는 *break* 문은 *else* 절을 실행하지 않고 루프를 종료한다. 첫 번째 스위트에서 실행되는 *continue* 문은 스위트의 나머지 부분을 건너뛰고 표현식의 검사로 돌아간다.

7.3 for 문

for 문은 (문자열, 튜플, 리스트 같은) 시퀀스 나 다른 이터러블 객체의 요소들을 이터레이트하는데 사용된다:

```
for_stmt  ::=  "for" target_list "in" expression_list ":" suite
               ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the expression_list. The suite is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the *else* clause, if present, is executed, and the loop terminates.

A *break* statement executed in the first suite terminates the loop without executing the *else* clause's suite. A *continue* statement executed in the first suite skips the rest of the suite and continues with the next item, or with

the *else* clause if there was no next item.

The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns a sequence of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `range(3)` returns the list `[0, 1, 2]`.

참고: There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

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

7.4 `try` 문

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

```
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
```

버전 2.5에서 변경: In previous versions of Python, *try*…*except*…*finally* did not work. *try*…*except* had to be nested in *try*…*finally*.

The *except* clause(s) specify one or more exception handlers. When no exception occurs in the *try* clause, no exception handler is executed. When an exception occurs in the *try* suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is《compatible》with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, or a tuple containing an item compatible with the exception.

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

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

When a matching except clause is found, the exception is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for

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

the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

Before an except clause's suite is executed, details about the exception are assigned to three variables in the `sys` module: `sys.exc_type` receives the object identifying the exception; `sys.exc_value` receives the exception's parameter; `sys.exc_traceback` receives a traceback object (see section 표준형 계층) identifying the point in the program where the exception occurred. These details are also available through the `sys.exc_info()` function, which returns a tuple `(exc_type, exc_value, exc_traceback)`. Use of the corresponding variables is deprecated in favor of this function, since their use is unsafe in a threaded program. As of Python 1.5, the variables are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional *else* clause is executed if the control flow leaves the *try* suite, no exception was raised, and no *return*, *continue*, or *break* statement was executed. Exceptions in the *else* clause are not handled by the preceding *except* clauses.

If *finally* is present, it specifies a 〈cleanup〉 handler. The *try* clause is executed, including any *except* and *else* clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The *finally* clause is executed. If there is a saved exception, it is re-raised at the end of the *finally* clause. If the *finally* clause raises another exception or executes a *return* or *break* statement, the saved exception is discarded:

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

finally 절을 실행하는 동안 예외 정보는 프로그램에 제공되지 않는다.

try…*finally* 문의 *try* 스위트에서 *return*, *break*, *continue* 문이 실행될 때, *finally* 절도 〈나가는 길에〉 실행된다. *finally* 절에서는 *continue* 문을 사용할 수 없다. (그 이유는 현재 구현에 있는 문제 때문이다 — 이 제약은 미래에 제거될 수 있다).

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

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

7.5 with 문

버전 2.5에 추가.

with 문은 블록의 실행을 컨텍스트 관리자 (*with* 문 컨텍스트 관리자 섹션을 보라) 가 정의한 메서드들로 감싸는 데 사용된다. 이것은 흔한 *try*…*except*…*finally* 사용 패턴을 편리하게 재사용할 수 있도록 캡슐화할 수 있도록 한다.

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

하나의 《item》 을 사용하는 *with* 문의 실행은 다음과 같이 진행된다:

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

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

5. 스위트가 실행된다.
6. The context manager's *__exit__()* method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to *__exit__()*. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the *__exit__()* method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the *with* statement.

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

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

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

는 다음과 동등하다

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

참고: In Python 2.5, the *with* statement is only allowed when the `with_statement` feature has been enabled. It is always enabled in Python 2.6.

버전 2.7에서 변경: 다중 컨텍스트 표현식의 지원

더 보기:

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

7.6 함수 정의

함수 정의는 사용자 정의 함수 객체 (표준형 계층 섹션을 보라) 를 정의한다:

```
decorated       ::=  decorators (classdef | funcdef)
decorators      ::=  decorator+
decorator       ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
funcdef         ::=  "def" funcname "(" [parameter_list] ")" ":" suite
dotted_name     ::=  identifier ("." identifier)*
parameter_list  ::=  (defparameter ",")*
                     ( "*" identifier ["," "**" identifier]
                     | "**" identifier
                     | defparameter [","] )
defparameter    ::=  parameter ["=" expression]
sublist         ::=  parameter ("," parameter)* [","]
parameter       ::=  identifier | "(" sublist ")"
funcname        ::=  identifier
```

함수 정의는 실행할 수 있는 문장이다. 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 사용된다.

함수 정의는 함수의 바디를 실행하지 않는다. 함수가 호출될 때 실행된다.[2]

A function definition may be wrapped by one or more *decorator* expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code:

```
@f1(arg)
@f2
def func(): pass
```

is equivalent to:

```
def func(): pass
func = f1(arg)(f2(func))
```

When one or more top-level *parameters* have the form *parameter* = *expression*, the function is said to have 《default parameter values.》 For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same 《pre-computed》 value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
```

(다음 페이지에 계속)

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

(이전 페이지에서 계속)

```
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section 호출. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form 《*identifier》 is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form 《**identifier》 is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section 람다*(Lambdas)*. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a 《*def*》 statement can be passed around or assigned to another name just like a function defined by a lambda expression. The 《*def*》 form is actually more powerful since it allows the execution of multiple statements.

Programmer's note: Functions are first-class objects. A 《def》 form executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section 이름과 연결*(binding)* for details.

7.7 클래스 정의

클래스 정의는 클래스 객체(표준형 계층 섹션을 보라)를 정의한다:

```
classdef     ::=   "class" classname [inheritance] ":" suite
inheritance  ::=   "(" [expression_list] ")"
classname    ::=   identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section 이름과 연결*(binding)*), using a newly created local namespace and the original global namespace. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved.[3] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

Programmer's note: Variables defined in the class definition are class variables; they are shared by all instances. To create instance variables, they can be set in a method with `self.name = value`. Both class and instance variables are accessible through the notation 《`self.name`》, and an instance variable hides a class variable with the same name when accessed in this way. Class variables can be used as defaults for instance variables, but using mutable values there can lead to unexpected results. For *new-style class*es, descriptors can be used to create instance variables with different implementation details.

Class definitions, like function definitions, may be wrapped by one or more *decorator* expressions. The evaluation rules for the decorator expressions are the same as for functions. The result must be a class object, which is then bound to the class name.

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

CHAPTER 8

최상위 요소들

파이썬 인터프리터는 여러 가지 출처로부터 입력을 얻을 수 있다: 표준 입력이나 프로그램 인자로 전달된 스크립트, 대화형으로 입력된 것, 모듈 소스 파일 등등. 이 장은 이 경우들에 사용되는 문법을 제공한다.

8.1 완전한 파이썬 프로그램

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `__builtin__` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

완전한 파이썬 프로그램의 문법은 다음 섹션에서 설명되는 파일 입력의 경우다.

인터프리터는 대화형으로 실행될 수도 있다; 이 경우, 완전한 프로그램을 읽어서 실행하지 않고, 한 번에 한 문장 (복합문도 가능하다) 씩 읽어서 실행한다. 초기 환경은 완전한 프로그램과 같다; 각 문장은 `__main__` 의 이름 공간에서 실행된다.

A complete program can be passed to the interpreter in three forms: with the `-c` *string* command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

8.2 파일 입력

비대화형 파일로부터 읽힌 모든 입력은 같은 형태를 취한다:

```
file_input  ::=  (NEWLINE | statement)*
```

이 문법은 다음과 같은 상황에서 사용된다:

- (파일이나 문자열로부터 온) 완전한 파이썬 프로그램을 파싱할 때;
- 모듈을 파싱할 때;
- when parsing a string passed to the *exec* statement;

8.3 대화형 입력

대화형 모드에서의 입력은 다음과 같은 문법 규칙을 사용한다:

```
interactive_input   ::=   [stmt_list] NEWLINE | compound_stmt NEWLINE
```

(최상위) 복합문은 대화형 모드에서 빈 줄을 붙여줘야 함에 유념해야 한다; 파서가 입력의 끝을 감지하는 데 필요하다.

8.4 표현식 입력

There are two forms of expression input. Both ignore leading whitespace. The string argument to `eval()` must have the following form:

```
eval_input   ::=   expression_list NEWLINE*
```

The input line read by `input()` must have the following form:

```
input_input   ::=   expression_list NEWLINE
```

Note: to read 〈raw〉 input line without interpretation, you can use the built-in function `raw_input()` or the `readline()` method of file objects.

CHAPTER 9

전체 문법 규격

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

```
# Grammar for Python

# Note:  Changing the grammar specified in this file will most likely
#        require corresponding changes in the parser module
#        (../Modules/parsermodule.c).  If you can't make the changes to
#        that module yourself, please co-ordinate the required changes
#        with someone who can; ask around on python-dev for help.  Fred
#        Drake <fdrake@acm.org> will probably be listening there.

# NOTE WELL: You should also follow all the steps listed in PEP 306,
# "How to Change Python's 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() and input() 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)
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [varargslist] ')'
varargslist: ((fpdef ['=' test] ',')*
              ('*' NAME [',' '**' NAME] | '**' NAME) |
              fpdef ['=' test] (',' fpdef ['=' test])* [','])
fpdef: NAME | '(' fplist ')'
fplist: fpdef (',' fpdef)* [',']
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal assignments, additional restrictions enforced by the interpreter
print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )
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 [',' test [',' test]]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
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)*
exec_stmt: 'exec' expr ['in' test [',' test]]
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated
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' | ',') test]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

# Backward compatibility cruft to support:
# [ x for x in lambda: True, lambda: False if x() ]
# even while also allowing:
# lambda x: 5 if x else 2
# (But not a mix of the two)
testlist_safe: old_test [(',' old_test)+ [',']]
old_test: or_test | old_lambdef
old_lambdef: 'lambda' [varargslist] ':' old_test
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
test: or_test ['if' or_test 'else' test] | lambdef
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)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
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 trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+)
listmaker: test ( list_for | (',' test)* [','] )
testlist_comp: test ( comp_for | (',' test)* [','] )
lambdef: 'lambda' [varargslist] ':' test
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [',')) |
                  (test (comp_for | (',' test)* [',')) )

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

arglist: (argument ',')* (argument [',']
                         |'*' test (',' argument)* [',' '**' test]
                         |'**' test)
# 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.
argument: test [comp_for] | test '=' test

list_iter: list_for | list_if
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
list_if: 'if' old_test [list_iter]

comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' old_test [comp_iter]

testlist1: test (',' test)*

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

yield_expr: 'yield' [testlist]
```

APPENDIX A

용어집

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

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 파이썬 2.x 코드를 파이썬 3.x 코드로 변환하려고 시도하는 도구인데, 소스를 파싱하고 파스 트리를 탐색해서 감지할 수 있는 대부분의 비호환성을 다룬다.

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

abstract base class (추상 베이스 클래스) Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with *magic methods*). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

argument (인자) A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- 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 인자다:

```
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 있다. 예를 들어, 다음과 같은 호출에서 `3` 과 `5` 는 모두 위치 인자다.

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

인자는 함수 바의 이름 붙은 지역 변수에 대입된다. 이 대입에 적용되는 규칙들에 대해서는 호출 섹션을 보세요. 문법적으로, 어떤 표현식이건 인자로 사용될 수 있다; 구해진 값이 지역 변수에 대입된다.

See also the *parameter* glossary entry and the FAQ question on the difference between arguments and parameters.

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

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

bytes-like object (바이트열류 객체) An object that supports the buffer protocol, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode (바이트 코드) Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This《intermediate language》is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

바이트 코드 명령어들의 목록은 dis 모듈 도큐멘테이션에 나온다.

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

classic class Any class which does not inherit from `object`. See *new-style class*. Classic classes have been removed in Python 3.

coercion (코어션) The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

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

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

CPython 파이썬 프로그래밍 언어의 규범적인 구현인데, python.org 에서 배포된다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어《CPython》이 사용된다.

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

데코레이터 문법은 단지 편의 문법일 뿐이다. 다음 두 함수 정의는 의미상으로 동등하다:

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

@staticmethod
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰인다. 데코레이터에 대한 더 자세한 내용은 함수 정의 와 클래스 정의 의 도큐멘테이션을 보면 된다.

descriptor (디스크립터) Any *new-style* object which defines the methods *__get__()*, *__set__()*, or *__delete__()*. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

디스크립터의 메서드들에 대한 자세한 내용은 디스크립터 구현하기 에 나온다.

dictionary (딕셔너리) An associative array, where arbitrary keys are mapped to values. The keys can be any object with *__hash__()* and *__eq__()* methods. Called a hash in Perl.

dictionary view (딕셔너리 뷰) The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

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

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

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 잡는다. 이 깔끔하고 빠른 스타일은 많은 *try* 와 *except* 문의 존재로 특징지어진다. 이 테크닉은 C와 같은 다른 많은 언어에서 자주 사용되는 *LBYL* 스타일과 대비된다.

expression (표현식) A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as *print* or *if*. Assignments are also statements, not expressions.

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

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 저장장치나 통신 장치 (예를 들어, 표준 입출력, 인-메모리 버퍼, 소켓, 파이프, 등등) 에 대한 액세스를 중계할 수 있다. 파일 객체는 파일류 객체 *(file-like objects)* 나 스트림 *(streams)* 이라고도 불린다.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

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

finder (파인더) An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See **PEP 302** for details.

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

function (함수) 호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 인자 가 전달될 수 있는데, 바디의 실행에 사용될 수 있다. 파라미터 와 메서드 와 함수 정의 섹션도 보세요.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

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

garbage collection (가비지 수거) The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

generator (제너레이터) A function which returns an iterator. It looks like a normal function except that it contains *yield* statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each *yield* temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression (제너레이터 표현식) 이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 *for* 표현식과 생략 가능한 *if* 표현식이 뒤에 붙는 일반 표현식 처럼 보인다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어낸다:

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

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 만든다. 인터프리터 전체를 로킹하는 것은 인터프리터를 다중스레드화하기 쉽게 만드는 대신, 다중 프로세서 기계가 제공하는 병렬성의 많은 부분을 희생한다.

하지만, 어떤 확장 모듈들은, 표준이나 제삼자 모두, 압축이나 해싱 같은 계산 집약적인 작업을 수행할 때는 GIL 을 반납하도록 설계되었다. 또한, I/O를 할 때는 항상 GIL 을 반납한다.

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

hashable (해시 가능) An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a *__hash__()* method), and can be compared to other objects (it needs an *__eq__()* or *__cmp__()* method). Hashable objects which compare equal must have the same hash value.

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

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

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

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

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

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 객체다.

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

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

iterable (이터러블) An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an *__iter__()* or *__getitem__()* method. Iterables can be used in a *for* loop and in many other places where a sequence is needed (`zip()`, `map()`, ⋯). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator (이터레이터) An object representing a stream of data. Repeated calls to the iterator's *next()* method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its *next()* method just raise `StopIteration` again. Iterators are required to have an *__iter__()* method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a *for* loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

typeiter 에 더 자세한 내용이 있다.

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

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a *lambda* expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

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` 이다.

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

다중 스레드 환경에서, LBYL 접근법은 《보기》와 《뛰기》 간에 경쟁 조건을 만들게 될 위험이 있다. 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 있다. 이런 이슈는 록이나 EAFP 접근법을 사용함으로써 해결될 수 있다.

list (리스트) A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is O(1).

list comprehension (리스트 컴프리헨션) A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The *if* clause is optional. If omitted, all elements in `range(256)` are processed.

loader (로더) An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

magic method An informal synonym for *special method*.

mapping (매핑) A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

metaclass (메타 클래스) 클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 딕셔너리, 베이스 클래스들의 목록을 만든다. 메타 클래스는 이 세 인자를 받아서 클래스를 만드는 책임을 진다. 대부분의 객체 지향형 프로그래밍 언어들은 기본 구현을 제공한다. 파이썬을 특별하게 만드는 것은 커스텀 메타 클래스를 만들 수 있다는 것이다. 대부분 사용자에게는 이 도구가 전혀 필요 없지만, 필요가 생길 때, 메타 클래스는 강력하고 우아한 해법을 제공한다. 어트리뷰트 액세스의 로깅 (logging), 스레드 안전성의 추가, 객체 생성 추적, 싱글톤 구현과 많은 다른 작업에 사용됐다.

클래스 생성 커스터마이제이션 에서 더 자세한 내용을 찾을 수 있다.

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

method resolution order (메서드 결정 순서) 메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 The Python 2.3 Method Resolution Order 를 보면 된다.

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

패키지 도 보세요.

MRO 메서드 결정 순서 를 보세요.

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

named tuple (네임드 튜플) 인덱싱할 수 있는 요소들을 이름 붙은 어트리뷰트로도 액세스할 수 있는 모든 튜플류 클래스 (예를 들어, `time.localtime()` 은 *year* 가 `t[0]` 처럼 인덱스로도, `t.tm_year` 처럼 어트리뷰트로도 액세스할 수 있는 튜플류 객체를 돌려준다.)

네임드 튜플은 `time.struct_time` 같은 내장형일 수도, 일반 클래스 정의로 만들 수도 있다. 모든 기능이 구현된 네임드 튜플을 팩토리 함수 `collections.namedtuple()` 로도 만들 수 있다. 마지막 접근법은 `Employee(name='jones', title='programmer')` 와 같은 스스로 문서로 만드는 repr 과 같은 확장 기능도 자동 제공한다.

namespace (이름 공간) The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

nested scope (중첩된 스코프) The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class (뉴스타일 클래스) Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and *`__getattribute__()`*.

More information can be found in *New-style and classic classes*.

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

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

parameter (파라미터) A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

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

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

- 위치-전용 *(positional-only)*: 위치로만 제공될 수 있는 인자를 지정한다. 파이썬은 위치-전용 파라미터를 정의하는 문법을 갖고 있지 않다. 하지만, 어떤 매장 함수들은 위치-전용 파라미터를 갖는다 (예를 들어, `abs()`).
- 가변-위치 *(var-positional)*: (다른 파라미터들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정한다. 이런 파라미터는 파라미터 이름에 `*` 를 앞에 붙여서 정의될 수 있다, 예를 들어 다음에서 *args*:

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

- 가변-키워드 *(var-keyword)*: (다른 파라미터들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정한다. 이런 파라미터는 파라미터 이름에 `**` 를 앞에 붙여서 정의될 수 있다, 예를 들어 위의 예 에서 *kwargs*.

파라미터는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있다.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the 함수 정의 section.

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

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

PEP 1 참조하세요.

positional argument (위치 인자) 인자 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 한다.

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

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

더 깔끔한, 파이썬다운 방법은 이렇다:

```
for piece in food:
    print piece
```

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

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence (시퀀스) An *iterable* which supports efficient element access using integer indices via the *__getitem__()* special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports *__getitem__()* and *__len__()*, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

slice (슬라이스) An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, *__getslice__()* and *__setslice__()*).

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

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

struct sequence (구조체 시퀀스) A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

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 있다. 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 있다.

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

universal newlines (유니버설 줄 넘김) A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `str.splitlines()` for an additional use.

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.

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

Zen of Python (파이썬 젠) 파이썬 디자인 원리와 철학들의 목록인데, 인어를 이해하고 사용하는 데 도움이 된다. 이 목록은 대화형 프롬프트에서 《`import this`》를 입력하면 보인다.

APPENDIX B

About these documents

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the reporting-bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!

APPENDIX C

History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (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.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 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 and above	2.1.1	2001-now	PSF	yes

참고: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.18

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   2.7.18 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 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 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 2.7.18 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
   2.7.18.

4. PSF is making Python 2.7.18 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 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, 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 2.7.18, 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

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

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

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

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


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

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
  ---------------------------------------------------------------------
 /                       Copyright (c) 1996.                           \
|          The Regents of the University of California.                 |
|                         All rights reserved.                          |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

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.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

C.3.6 Cookie management

The `Cookie` module contains the following notice:

```
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.3.7 Execution tracing

The `trace` module contains the following notice:

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

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

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

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

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


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

C.3.8 UUencode and UUdecode functions

The `uu` module contains the following notice:

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

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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 XML Remote Procedure Calls

The `xmlrpclib` module contains the following notice:

```
    The XML-RPC client interface is

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

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

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

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

C.3.10 test_epoll

The `test_epoll` contains the following notice:

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

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Select kqueue

The `select` and contains the following notice for the kqueue interface:

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

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

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

C.3.12 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 * 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.3.13 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   * 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * 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.3.14 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

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

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

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

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

C.3.15 libffi

The `_ctypes` extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

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

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

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

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

C.3.16 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2010 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
```

APPENDIX D

저작권

파이썬과 이 도큐멘테이션은:

전체 라이센스 및 사용 권한 정보는 *History and License* 에서 제공한다.

Non-alphabetical

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z