

---

# Python Tutorial

릴리스 3.13.3

Guido van Rossum and the Python development team

4월 27, 2025



<b>1</b>	<b>입맛 돋우기</b>	<b>3</b>
<b>2</b>	<b>파이썬 인터프리터 사용하기</b>	<b>5</b>
2.1	인터프리터 실행하기	5
2.2	인터프리터와 환경	6
<b>3</b>	<b>파이썬의 간략한 소개</b>	<b>9</b>
3.1	파이썬을 계산기로 사용하기	9
3.2	프로그래밍으로의 첫걸음	16
<b>4</b>	<b>기타 제어 흐름 도구</b>	<b>19</b>
4.1	if 문	19
4.2	for 문	19
4.3	range() 함수	20
4.4	break and continue Statements	21
4.5	else Clauses on Loops	22
4.6	pass 문	22
4.7	match Statements	23
4.8	함수 정의하기	25
4.9	함수 정의 더 보기	27
4.10	막간극: 코딩 스타일	34
<b>5</b>	<b>자료 구조</b>	<b>35</b>
5.1	리스트 더 보기	35
5.2	del 문	39
5.3	튜플과 시퀀스	40
5.4	집합	41
5.5	딕셔너리	41
5.6	루프 테크닉	42
5.7	조건 더 보기	44
5.8	시퀀스와 다른 형들을 비교하기	44
<b>6</b>	<b>모듈</b>	<b>47</b>
6.1	모듈 더 보기	48
6.2	표준 모듈들	50
6.3	dir() 함수	51
6.4	패키지	52
<b>7</b>	<b>입력과 출력</b>	<b>57</b>
7.1	장식적인 출력 포매팅	57
7.2	파일을 읽고 쓰기	61

<b>8</b>	<b>에러와 예외</b>	<b>65</b>
8.1	문법 에러	65
8.2	예외	65
8.3	예외 처리하기	66
8.4	예외 일으키기	68
8.5	예외 연쇄	69
8.6	사용자 정의 예외	70
8.7	뒷정리 동작 정의하기	70
8.8	미리 정의된 뒷정리 동작들	72
8.9	Raising and Handling Multiple Unrelated Exceptions	72
8.10	Enriching Exceptions with Notes	74
<b>9</b>	<b>클래스</b>	<b>77</b>
9.1	이름과 객체에 관한 한마디	77
9.2	파이썬 스코프와 이름 공간	78
9.3	클래스와의 첫 만남	80
9.4	기타 주의사항들	83
9.5	상속	84
9.6	비공개 변수	85
9.7	잡동사니	86
9.8	이터레이터	87
9.9	제너레이터	88
9.10	제너레이터 표현식	89
<b>10</b>	<b>표준 라이브러리 둘러보기</b>	<b>91</b>
10.1	운영 체제 인터페이스	91
10.2	파일 와일드카드	91
10.3	명령행 인자	92
10.4	에러 출력 리디렉션과 프로그램 종료	92
10.5	문자열 패턴 매칭	92
10.6	수학	93
10.7	인터넷 액세스	93
10.8	날짜와 시간	94
10.9	데이터 압축	94
10.10	성능 측정	94
10.11	품질 관리	95
10.12	배터리가 포함됩니다	95
<b>11</b>	<b>표준 라이브러리 둘러보기 — 2부</b>	<b>97</b>
11.1	출력 포매팅	97
11.2	템플릿	98
11.3	바이너리 데이터 레코드 배치 작업	99
11.4	다중 스테딩	99
11.5	로깅	100
11.6	약한 참조	101
11.7	리스트 작업 도구	101
11.8	Decimal Floating-Point Arithmetic	102
<b>12</b>	<b>가상 환경 및 패키지</b>	<b>105</b>
12.1	소개	105
12.2	가상 환경 만들기	105
12.3	pip로 패키지 관리하기	106
<b>13</b>	<b>이제 뭘 하지?</b>	<b>109</b>
<b>14</b>	<b>대화형 입력 편집 및 히스토리 치환</b>	<b>111</b>
14.1	탭 완성 및 히스토리 편집	111
14.2	대화형 인터프리터 대안	111

<b>15 Floating-Point Arithmetic: Issues and Limitations</b>	<b>113</b>
15.1 표현 오류	116
<b>16 부록</b>	<b>119</b>
16.1 대화형 모드	119
<b>A 용어집</b>	<b>121</b>
<b>B 이 설명서에 관하여</b>	<b>139</b>
B.1 파이썬 설명서의 공헌자들	139
<b>C 역사와 라이선스</b>	<b>141</b>
C.1 소프트웨어의 역사	141
C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관	142
C.3 포함된 소프트웨어에 대한 라이선스 및 승인	145
<b>D 저작권</b>	<b>161</b>
색인	<b>163</b>



파이썬은 배우기 쉬우면서도 강력한 프로그래밍 언어입니다. 효율적인 자료 구조와 객체 지향 프로그래밍에 대한 간단하고도 효과적인 접근법을 제공합니다. 파이썬은 우아한 문법과 동적 타이핑(typing)을 지원하는 인터프리터 언어로서, 대부분 플랫폼과 다양한 문제 영역에서 스크립트 작성과 빠른 응용 프로그램 개발에 이상적인 환경을 제공합니다.

파이썬 인터프리터와 풍부한 표준 라이브러리는 소스나 바이너리 형태로 파이썬 웹 사이트, <https://www.python.org/>, 에서 무료로 제공되고, 자유롭게 배포할 수 있습니다. 같은 사이트는 제삼자들이 무료로 제공하는 확장 모듈, 프로그램, 도구, 문서들의 배포판이나 링크를 포함합니다.

파이썬 인터프리터는 C 나 C++ (또는 C에서 호출 가능한 다른 언어들)로 구현된 새 함수나 자료 구조를 쉽게 추가할 수 있습니다. 파이썬은 고객화 가능한 응용 프로그램을 위한 확장 언어로도 적합합니다.

이 학습서는 파이썬 언어와 시스템의 기본 개념과 기능들을 격식 없이 소개합니다. 파이썬 인터프리터를 직접 만져볼 수 있도록 돕지만, 모든 예제가 독립적이기 때문에 오프라인에서 읽기에도 적합합니다.

표준 객체들과 모듈들에 대한 설명은 `library-index` 를 보세요. `reference-index` 는 언어에 대한 좀 더 형식적인 정의를 제공합니다. C 나 C++ 로 확장하려면 `extending-index` 와 `c-api-index` 를 읽으세요. 파이썬을 깊이 있게 다룬 책들도 많습니다.

이 자습서는 포괄적이려고 시도하지 않습니다. 모든 기능을 다루지는 않는데, 심지어 자주 사용되는 기능조차도 그렇습니다. 대신에, 파이썬의 가장 주목할만한 기능들을 소개하고, 언어의 맛과 스타일에 대한 전체적인 인상을 제공합니다. 이 학습서를 읽은 후에는 파이썬 모듈과 프로그램을 작성할 수 있고, `library-index` 에 기술된 다양한 파이썬 라이브러리 모듈들에 대해 학습할 수 있는 준비가 될 것입니다.

[용어집](#) 또한 훑어볼 만한 가치가 있습니다.





## 입맛 돋우기

여러분이 컴퓨터를 많이 사용한다면, 결국 자동화하고 싶은 작업을 발견하게 됩니다. 예를 들어, 많은 텍스트 파일들을 검색-수정하고 싶거나, 사진 파일들을 복잡한 방법으로 이름을 바꾸거나 재배치하고 싶을 수 있습니다. 어쩌면 자그마한 자신만의 데이터베이스나 GUI 응용 프로그램, 또는 간단한 게임을 만들고 싶을 것입니다.

만약 여러분이 전문 소프트웨어 개발자라면, 여러 C/C++/Java 라이브러리들을 갖고 작업해야만 할 수 있는데, 일반적인 코드작성/컴파일/테스트/재컴파일 순환이 너무 느리다는 것을 깨닫게 됩니다. 어쩌면 그 라이브러리들을 위한 테스트 스위트를 작성하다가, 테스트 코드 작성에 따분해하는 자신을 발견하게 됩니다. 또는 확장 언어를 사용하는 프로그램을 작성했는데, 완전히 새로운 언어 전체를 설계하고 구현하고 싶지 않을 수 있습니다.

파이썬은 바로 여러분을 위한 언어입니다.

여러분은 이런 작업들을 유닉스 셸 스크립트나 윈도우 배치 파일을 작성해서 해결할 수도 있습니다. 하지만 셸 스크립트는 파일을 이리저리 옮기거나 텍스트 데이터를 변경하는 데는 쓸모 있지만, GUI 응용 프로그램이나 게임을 만드는 데는 적합하지 않습니다. C/C++/Java 프로그램을 작성할 수도 있지만, 첫 초벌 프로그램을 만드는 데도 막대한 개발 시간이 들어갑니다. 파이썬은 사용하기에 더 간단하고, 윈도우, 맥 OS, 유닉스 운영체제에서 사용할 수 있으며, 더 빨리 작업을 완료할 수 있도록 합니다.

파이썬은 사용이 간단하지만, 제대로 갖춰진 프로그래밍 언어인데, 셸 스크립트나 배치 파일보다 더 많은 구조를 제공하고 커다란 프로그램을 위한 지원을 제공합니다. 반면에, 파이썬은 C보다 훨씬 많은 예러 검사를 제공하고, 유연한 배열과 딕셔너리같은 고수준의 자료형들을 내장하고 있습니다. 더 일반적인 자료형들 때문에 Awk 나 Perl보다도 더 많은 문제영역에 쓸모가 있는데, 그러면서도 여전히 많은 것들이 적어도 이들 언어를 사용하는 것만큼 파이썬에서도 쉽게 해결할 수 있습니다.

파이썬은 여러분의 프로그램을 여러 모듈로 나눌 수 있도록 하는데, 각 모듈은 다른 파이썬 프로그램에서 재사용할 수 있습니다. 대규모의 표준 모듈들이 따라오는데 여러분의 프로그램 기초로 사용하거나 파이썬 프로그래밍을 배우기 위한 예제로 활용할 수 있습니다. 이 모듈에는 파일 입출력, 시스템 호출, 소켓들이 포함되는데, 심지어 Tk 와 같은 GUI 도구상자에 대한 인터페이스도 들어있습니다.

파이썬은 인터프리터 언어입니다. 컴파일과 링크 단계가 필요 없으므로 개발 시간을 상당히 단축해줍니다. 인터프리터는 대화형으로 사용할 수 있어서, 언어의 기능을 실험하거나, 쓰고 버릴 프로그램을 만들거나, 바닥부터 프로그램을 만들어가는 동안 함수들을 테스트하기 쉽습니다. 간편한 탁상용 계산기이기도 합니다.

파이썬은 간결하고 읽기 쉽게 프로그램을 작성할 수 있도록 합니다. 파이썬 프로그램은 여러 가지 이유로 같은 기능의 C, C++, Java 프로그램들에 비교해 간결합니다:

- 고수준의 자료형 때문에 복잡한 연산을 한 문장으로 표현할 수 있습니다;
- 문장의 묶음은 괄호 대신에 들여쓰기를 통해 이루어집니다;

- 변수나 인자의 선언이 필요 없다.

파이썬은 확장 가능 하다: C로 프로그램하는 법을 안다면, 인터프리터에 새로운 내장 함수나 자료형을 추가해서, 핵심 연산을 최대 속도로 수행하거나 바이너리 형태로만 제공되는 라이브러리(가령 업체가 제공하는 그래픽스 라이브러리)에 파이썬 프로그램을 연결할 수 있습니다. 진짜 파이썬에 매료되었다면, C로 만든 응용 프로그램에 파이썬 인터프리터를 연결하여 그 응용 프로그램의 확장이나 명령 언어로 사용할 수 있습니다.

파이썬이라는 이름은 “Monty Python’s Flying Circus”라는 BBC 쇼에서 따온 것이고, 파충류와는 아무런 관련이 없습니다. 문서에서 Monty Python의 농담을 인용하는 것은 허락된 것일 뿐만 아니라, 권장되고 있습니다.

이제 여러분은 파이썬에 한껏 흥분한 상태고 좀 더 자세히 들여다보길 원할 것입니다. 언어를 배우는 가장 좋은 방법은 사용하는 것이기 때문에, 이 학습서를 읽으면서 직접 파이썬 인터프리터를 만져볼 것을 권합니다.

다음 장에서, 인터프리터를 사용하는 방법을 설명합니다. 이것은 약간 지루할 수도 있는 정보지만, 이후에 나오는 예제들을 실행하기 위해서는 꼭 필요합니다.

자습서의 나머지는 파이썬 언어와 시스템의 여러 기능을 예제를 통해 소개합니다. 간단한 표현식, 문장, 자료형에서 출발해서 함수와 모듈을 거쳐, 마지막으로 예외와 사용자 정의 클래스와 같은 고급 개념들을 다룹니다.

## 파이썬 인터프리터 사용하기

## 2.1 인터프리터 실행하기

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.13
```

을 셸에 입력해서 실행할 수 있습니다.<sup>1</sup> 인터프리터가 위치하는 디렉터리의 선택은 설치 옵션이기 때문에, 다른 장소도 가능합니다; 주변의 파이썬 전문가나 시스템 관리자에게 확인할 필요가 있습니다. (예를 들어, `/usr/local/python` 도 널리 사용되는 위치입니다.)

On Windows machines where you have installed Python from the Microsoft Store, the `python3.13` command will be available. If you have the `py.exe` launcher installed, you can use the `py` command. See `setting-envvars` for other ways to launch Python.

기본 프롬프트에서 EOF(end-of-file) 문자(유닉스에서는 `Control-D`, 윈도우에서는 `Control-Z`)를 입력하면 인터프리터가 종료하고, 종료 상태 코드는 0 이 됩니다. 이 방법이 통하지 않는다면 `quit()` 명령을 입력해서 인터프리터를 종료시킬 수 있습니다.

인터프리터는 **GNU Readline** 라이브러리를 지원하는 시스템에서 줄 편집 기능으로 대화형 편집, 히스토리 치환, 코드 완성 등을 제공합니다. 아마도 명령행 편집이 제공되는지 확인하는 가장 빠른 방법은 첫 프롬프트에서 `Control-P` 를 입력하는 것입니다. 뽁 하는 소리가 난다면 명령행 편집이 지원되고 있습니다; 입력 키에 대한 소개는 부록 **대화형 입력 편집 및 히스토리 치환** 을 보세요. 아무런 반응도 없거나 `^P` 가 출력된다면 명령행 편집이 제공되지 않는 것입니다; 현재 줄에서 문자를 지우기 위해 백스페이스를 사용할 수 있는 것이 전부입니다.

인터프리터는 어느 정도 유닉스 셸처럼 동작합니다: `tty` 장치에 표준 입력이 연결된 상태로 실행되면, 대화형으로 명령을 읽고 실행합니다; 파일명을 인자로 주거나 파일을 표준입력으로 연결한 상태로 실행되면 스크립트를 읽고 실행합니다.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

몇몇 파이썬 모듈들은 스크립트로도 쓸모가 있습니다. `python -m module [arg] ...` 로 실행할 수 있는데, 마치 *module* 모듈 소스 파일의 경로명을 명령행에 입력한 것처럼 실행되게 됩니다.

<sup>1</sup> 유닉스에서, 파이썬 3.x 인터프리터는 보통 `python` 이라는 이름의 실행 파일로 설치되지 않는데, 동시에 설치되는 파이썬 2.x 실행 파일과 충돌하지 않도록 하기 위해입니다.

스크립트 파일이 사용될 때, 때로 스크립트를 실행한 후에 대화형 모드로 들어가는 것이 편리할 때가 있습니다. 스크립트 앞에 `-i` 를 전달하면 됩니다.

모든 명령행 옵션은 `using-on-general` 에서 찾을 수 있습니다.

## 2.1.1 인자 전달

스크립트 이름과 추가의 인자들이 인터프리터로 전달될 때, 문자열의 목록으로 변환된 후 `sys` 모듈의 `argv` 변수에 저장됩니다. `import sys` 를 사용해서 이 목록에 접근할 수 있습니다. 목록의 길이는 최소한 1 이고, 스크립트도 추가의 인자도 없는 경우로, `sys.argv[0]` 은 빈 문자열입니다. 스크립트 이름을 '-' (표준 입력을 뜻한다) 로 주면 `sys.argv[0]` 는 '-' 가 됩니다. `-c command` 가 사용되면 `sys.argv[0]` 는 '-c' 로 설정됩니다. `-m module` 이 사용되면 `sys.argv[0]` 는 모듈의 절대 경로명이 됩니다. `-c command` 나 `-m module` 뒤에 오는 옵션들은 파이썬 인터프리터가 소모하지 않고 명령이나 모듈이 처리하도록 `sys.argv` 로 전달됩니다.

## 2.1.2 대화형 모드

명령을 `tty` 에서 읽을 때, 인터프리터가 대화형 모드로 동작한다고 말합니다. 이 모드에서는 기본 프롬프트를 표시해서 다음 명령을 요청하는데, 보통 세 개의 ...보다 크다 기호입니다 (`>>>`); 한 줄로 끝나지 않고 이어지는 줄의 입력을 요청할 때는 보조 프롬프트가 사용되는데, 기본적으로 세 개의 점입니다 (...). 인터프리터는 첫 번째 프롬프트를 인쇄하기 전에 버전 번호와 저작권 공지를 포함하는 환영 메시지를 출력합니다.

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

이어지는 줄은 여러 줄로 구성된 구조물을 입력할 때 필요합니다. 예를 들자면, 이런 식의 `if` 문이 가능합니다:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

대화형 모드에 대해 더 알고 싶다면, [대화형 모드](#) 를 보세요.

## 2.2 인터프리터와 환경

### 2.2.1 소스 코드 인코딩

기본적으로, 파이썬 소스 파일들은 UTF-8으로 인코딩 된 것으로 취급됩니다. 이 인코딩에서는 대부분 언어에서 사용되는 문자들을 문자열 상수, 식별자, 주석 등에서 함께 사용할 수 있습니다. (하지만 표준 라이브러리는 오직 ASCII 문자만 식별자로 사용하고 있는데, 범용 코드에서는 이 관례를 따르는 것이 좋습니다.) 이 문자들을 모두 올바르게 표시하기 위해서는 편집기가 파일이 UTF-8임을 인식해야 하고, 이 파일에 포함된 모든 문자를 지원할 수 있는 글꼴을 사용해야 합니다.

인코딩을 기본값 외의 것으로 선언하려면, 파일의 첫 줄에 특별한 형태의 주석 문을 추가해야 합니다. 문법은 이렇습니다:

```
# -*- coding: encoding -*-
```

`encoding` 은 파이썬이 지원하는 코덱 (codecs) 중 하나여야 합니다.

예를 들어, Windows-1252 인코딩을 사용하도록 선언하려면, 소스 코드 파일의 첫 줄은 이렇게 되어야 합니다:

```
# -*- coding: cp1252 -*-
```

첫 줄 규칙의 한가지 예외는 소스 코드가 유닉스 “셔뱅 (shebang)” 줄로 시작하는 경우입니다. 이 경우에, 인코딩 선언은 두 번째 줄에 들어갑니다. 예를 들어:

```
#!/usr/bin/env python3  
# -*- coding: cp1252 -*-
```



---

## 파이썬의 간략한 소개

---

다음에 나올 예에서, 입력과 출력은 프롬프트(> 와 ...)의 존재 여부로 구분됩니다: 예제를 실행하기 위해서는 프롬프트가 나올 때 프롬프트 뒤에 오는 모든 것들을 입력해야 합니다; 프롬프트로 시작하지 않는 줄들은 인터프리터가 출력하는 것들입니다. 예에서 보조 프롬프트 외에 아무것도 없는 줄은 빈 줄을 입력해야 한다는 뜻임에 주의하세요; 여러 줄로 구성된 명령을 끝내는 방법입니다.

이 설명서에 나오는 많은 예는 (대화형 프롬프트에서 입력되는 것들조차도) 주석을 포함하고 있습니다. 파이썬에서 주석은 해시 문자, #, 로 시작하고 줄의 끝까지 이어집니다. 주석은 줄의 처음에서 시작할 수도 있고, 공백이나 코드 뒤에 나올 수도 있습니다. 하지만 문자열 리터럴 안에는 들어갈 수 없습니다. 문자열 리터럴 안에 등장하는 해시 문자는 주석이 아니라 해시 문자일 뿐입니다. 주석은 코드의 의미를 정확히 전달하기 위한 것이고, 파이썬이 해석하지 않는 만큼, 예를 입력할 때는 생략해도 됩니다.

몇 가지 예를 듭니다:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

### 3.1 파이썬을 계산기로 사용하기

몇 가지 간단한 파이썬 명령을 사용해봅시다. 인터프리터를 실행하고 기본 프롬프트, >>>, 를 기다리세요. (얼마 걸리지 않아야 합니다.)

#### 3.1.1 숫자

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, \* and / can be used to perform arithmetic; parentheses (()) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 8 / 5 # division always returns a floating-point number
1.6
```

정수 (예를 들어 2, 4, 20)는 int 형입니다. 소수부가 있는 것들 (예를 들어 5.0, 1.6)은 float 형입니다. 이 자습서 뒤에서 숫자 형들에 대해 더 자세히 살펴볼 예정입니다.

Division (/) always returns a float. To do *floor division* and get an integer result you can use the // operator; to calculate the remainder you can use %:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17
```

파이썬에서는 거듭제곱을 계산할 때 \*\* 연산자를 사용합니다<sup>1</sup>:

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

변수에 값을 대입할 때는 등호(=)를 사용합니다. 이 경우 다음 대화형 프롬프트 전에 표시되는 출력은 없습니다:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

변수가 “정의되어” 있지 않을 때 (값을 대입하지 않았을 때) 사용하려고 시도하는 것은 에러를 일으킵니다:

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

실수를 본격적으로 지원합니다; 서로 다른 형의 피연산자를 갖는 연산자는 정수 피연산자를 실수로 변환합니다:

```
>>> 4 * 3.75 - 1
14.0
```

대화형 모드에서는, 마지막에 인쇄된 표현식은 변수 \_ 에 대입됩니다. 이것은 파이썬을 탁상용 계산기로 사용할 때, 계산을 이어 가기가 좀 더 쉬워짐을 의미합니다. 예를 들어:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
```

(다음 페이지에 계속)

<sup>1</sup> \*\* 가 - 보다 우선순위가 높으므로, -3\*\*2 는 -(3\*\*2) 로 해석되어서 결과는 -9 가 됩니다. 9 를 얻고 싶으면 (-3)\*\*2 를 사용할 수 있습니다.



(이전 페이지에서 계속)

```
113.0625
>>> round(_, 2)
113.06
```

이 변수는 사용자로서는 읽기만 가능한 것처럼 취급되어야 합니다. 값을 직접 대입하지 마세요 — 만약 그렇게 한다면 같은 이름의 지역 변수를 새로 만드는 것이 되는데, 내장 변수의 마술 같은 동작을 차단하는 결과를 낳습니다.

`int` 와 `float` 에 더해, 파이썬은 `Decimal` 이나 `Fraction` 등의 다른 형의 숫자들도 지원합니다. 파이썬은 복소수 에 대한 지원도 내장하고 있는데, 허수부를 가리키는데 `j` 나 `J` 접미사를 사용합니다 (예를 들어 `3+5j`).

### 3.1.2 Text

Python can manipulate text (represented by type `str`, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result<sup>2</sup>.

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use \' to escape the single quote...
"doesn't"
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> 'Isn\'t," they said.'
'Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces
↵new line
First line.
Second line.
```

`\` 뒤에 나오는 문자가 특수 문자로 취급되게 하고 싶지 않다면, 첫 따옴표 앞에 `r` 을 붙여서 날 문자열 (*raw string*) 을 만들 수 있습니다:

<sup>2</sup> 다른 언어들과는 달리, `\n` 과 같은 특수 문자들은 작은따옴표(`'...'`)와 큰따옴표(`"..."`)에서 같은 의미가 있습니다. 둘 간의 유일한 차이는 작은따옴표 안에서 `"` 를 이스케이핑할 필요가 없고 (하지만 `\'` 는 이스케이핑 시켜야 합니다), 그 역도 성립한다는 것입니다.

```
>>> print('C:\some\name') # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see the FAQ entry for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it's possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...     -h                        Display this usage message
...     -H hostname              Hostname to connect to
... """)
Usage: thingy [OPTIONS]
    -h                        Display this usage message
    -H hostname              Hostname to connect to

>>>
```

문자열은 `+` 연산자로 이어붙이고, `*` 연산자로 반복시킬 수 있습니다:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

두 개 이상의 문자열 리터럴 (즉, 따옴표로 둘러싸인 것들) 가 연속해서 나타나면 자동으로 이어 붙여집니다.

```
>>> 'Py' 'thon'
'Python'
```

이 기능은 긴 문자열을 쪼개고자 할 때 특별히 쓸모 있습니다:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

이것은 오직 두 개의 리터럴에만 적용될 뿐 변수나 표현식에는 해당하지 않습니다:

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
File "<stdin>", line 1
    prefix 'thon'
    ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
File "<stdin>", line 1
    ('un' * 3) 'ium'
    ^^^^^^
SyntaxError: invalid syntax
```

변수들끼리 혹은 변수와 문자열 리터럴을 이어붙이려면 `+` 를 사용해야 합니다

```
>>> prefix + 'thon'
'Python'
```

문자열은 인덱스 (서브 스크립트) 될 수 있습니다. 첫 번째 문자가 인덱스 0에 대응됩니다. 문자를 위한 별도의 형은 없습니다; 단순히 길이가 1인 문자열입니다:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

인덱스는 음수가 될 수도 있는데, 끝에서부터 셉니다:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

-0은 0과 같으므로, 음의 인덱스는 -1에서 시작한다는 것에 주목하세요.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

슬라이스 인덱스는 편리한 기본값을 갖고 있습니다; 첫 번째 인덱스를 생략하면 기본값 0이 사용되고, 두 번째 인덱스가 생략되면 기본값으로 슬라이싱 되는 문자열의 길이가 사용됩니다.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

시작 위치의 문자는 항상 포함되는 반면, 종료 위치의 문자는 항상 포함되지 않는 것에 주의하세요. 이 때문에 `s[:i] + s[i:]`는 항상 `s`와 같아집니다

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

슬라이스가 동작하는 방법을 기억하는 한 가지 방법은 인덱스가 문자들 사이의 위치를 가리킨다고 생각하는 것입니다. 첫 번째 문자의 왼쪽 경계가 0입니다.  $n$  개의 문자들로 구성된 문자열의 오른쪽 끝 경계는 인덱스  $n$ 이 됩니다, 예를 들어:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

첫 번째 숫자 행은 인덱스 0...6의 위치를 보여주고; 두 번째 행은 대응하는 음의 인덱스들을 보여줍니다.  $i$ 에서  $j$  범위의 슬라이스는  $i$ 와  $j$ 로 번호 붙여진 경계 사이의 문자들로 구성됩니다.

음이 아닌 인덱스들의 경우, 두 인덱스 모두 범위 내에 있다면 슬라이스의 길이는 인덱스 간의 차이입니다. 예를 들어 `word[1:3]`의 길이는 2입니다.

너무 큰 값을 인덱스로 사용하는 것은 에러입니다:

```
>>> word[42] # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

하지만, 범위를 벗어나는 슬라이스 인덱스는 슬라이싱할 때 부드럽게 처리됩니다:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

파이썬 문자열은 변경할 수 없다 — 불변 이라고 합니다. 그래서 문자열의 인덱스로 참조한 위치에 대입하려고 하면 에러를 일으킵니다:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

다른 문자열이 필요하다면, 새로 만들어야 합니다:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

내장 함수 `len()`은 문자열의 길이를 돌려줍니다:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

## ➡ 더 보기

### textseq

문자열은 시퀀스 형의 일종이고, 시퀀스가 지원하는 공통 연산들이 지원됩니다.

### string-methods

문자열은 기본적인 변환과 검색을 위한 여러 가지 메서드들을 지원합니다.

### f-strings

내장된 표현식을 갖는 문자열 리터럴

### formatstrings

`str.format()`으로 문자열을 포맷하는 방법에 대한 정보.

**old-string-formatting**

이곳에서 문자열을 % 연산자 왼쪽에 사용하는 예전 방식의 포매팅에 관해 좀 더 상세하게 설명하고 있습니다.

**3.1.3 리스트**

파이썬은 다른 값들을 덩어리로 묶는데 사용되는 여러 가지 컴파운드 (*compound*) 자료형을 알고 있습니다. 가장 융통성이 있는 것은 리스트 인데, 대괄호 사이에 쉼표로 구분된 값(항목)들의 목록으로 표현될 수 있습니다. 리스트는 서로 다른 형의 항목들을 포함할 수 있지만, 항목들이 모두 같은 형인 경우가 많습니다.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

문자열(그리고, 다른 모든 내장 시퀀스 형들)처럼 리스트는 인덱싱하고 슬라이싱할 수 있습니다:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

리스트는 이어붙이기 같은 연산도 지원합니다:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

불변 인 문자열과는 달리, 리스트는 가변 입니다. 즉 내용을 변경할 수 있습니다:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` *method* (we will see more about *methods* later):

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it.:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append("Alpha")
>>> rgb
["Red", "Green", "Blue", "Alpha"]
```

모든 슬라이스 연산은 요청한 항목들을 포함하는 새 리스트를 돌려줍니다. 이는 다음과 같은 슬라이스가 리스트의 새로운 얇은 복사본을 돌려준다는 뜻입니다:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
>>> rgba
["Red", "Green", "Blue", "Alph"]
```

슬라이스에 대입하는 것도 가능한데, 리스트의 길이를 변경할 수 있고, 모든 항목을 삭제할 수조차 있습니다:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

내장 함수 `len()` 은 리스트에도 적용됩니다:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

리스트를 중첩할 수도 있습니다. (다른 리스트를 포함하는 리스트를 만듭니다). 예를 들어:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

## 3.2 프로그래밍으로의 첫걸음

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the [Fibonacci series](#) as follows:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
... 
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
0
1
1
2
3
5
8
```

이 예는 몇 가지 새로운 기능을 소개하고 있습니다.

- 첫 줄은 다중 대입 을 포함하고 있습니다: 변수 `a` 와 `b` 에 동시에 값 0과1이 대입됩니다. 마지막 줄에서 다시 사용되는데, 대입이 어느 하나라도 이루어지기 전에 우변의 표현식들이 모두 계산됩니다. 우변의 표현식은 왼쪽부터 오른쪽으로 가면서 순서대로 계산됩니다.
- `while` 루프는 조건(여기서는: `a < 10`)이 참인 동안 실행됩니다. `C`와 마찬가지로 파이썬에서 0 이 아닌 모든 정수는 참이고, 0은 거짓입니다. 조건은 문자열이나 리스트 (사실 모든 종류의 시퀀스) 가 될 수도 있는데 길이가 0 이 아닌 것은 모두 참이고, 빈 시퀀스는 거짓입니다. 이 예에서 사용한 검사는 간단한 비교입니다. 표준 비교 연산자는 `C`와 같은 방식으로 표현됩니다: `<` (작다), `>` (크다), `==` (같다), `<=` (작거나 같다), `>=` (크거나 같다), `!=` (다르다).
- 루프의 바디 (*body*) 는 들여쓰기 됩니다. 들여쓰기는 파이썬에서 문장을 덩어리로 묶는 방법입니다. 대화형 프롬프트에서 각각 들여 쓰는 줄에서 탭 (`tab`) 이나 공백 (`space`) 을 입력해야 합니다. 실제적으로는 텍스트 편집기를 사용해서 좀 더 복잡한 파이썬 코드를 준비하게 됩니다; 웬만한 텍스트 편집기들은 자동 들여쓰기 기능을 제공합니다. 복합문을 대화형으로 입력할 때는 끝을 알리기 위해 빈 줄을 입력해야 합니다. (해석기가 언제 마지막 줄을 입력할지 짐작할 수 없기 때문입니다.) 같은 블록에 포함되는 모든 줄은 같은 양만큼 들여쓰기 되어야 함에 주의하세요.
- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating-point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

키워드 인자 `end` 는 출력 끝에 포함되는 개행문자를 제거하거나 출력을 다른 문자열로 끝나게 하고 싶을 때 사용됩니다:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=', ')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```





## 기타 제어 흐름 도구

As well as the `while` statement just introduced, Python uses a few more that we will encounter in this chapter.

### 4.1 `if` 문

아마도 가장 잘 알려진 문장 형은 `if` 문일 것입니다. 예를 들어:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

없거나 여러 개의 `elif` 부가 있을 수 있고, `else` 부는 선택적입니다. 키워드 ‘`elif`’ 는 ‘`else if`’ 의 줄임 표현인데, 과도한 들여쓰기를 피하는 데 유용합니다. `if ... elif ... elif ...` 시퀀스는 다른 언어들에서 발견되는 `switch` 나 `case` 문을 대신합니다.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [match Statements](#).

### 4.2 `for` 문

파이썬에서 `for` 문은 C 나 파스칼에서 사용하던 것과 약간 다릅니다. (파스칼처럼) 항상 숫자의 산술적인 진행을 통해 이터레이션 하거나, (C처럼) 사용자가 이터레이션 단계와 중지 조건을 정의할 수 있도록 하는 대신, 파이썬의 `for` 문은 임의의 시퀀스 (리스트나 문자열)의 항목들을 그 시퀀스에 들어있는 순서대로 이터레이션 합니다. 예를 들어 (말장난이 아니라):

```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

컬렉션을 이터레이트 하는 동안 같은 컬렉션을 수정하는 코드는 올바르게 동작하도록 만들기 힘듭니다. 대신, 보통 컬렉션의 복사본으로 루프를 만들거나 새 컬렉션을 만드는 것이 더 간단합니다:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', 'bug': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

## 4.3 range() 함수

숫자들의 시퀀스로 이터레이트할 필요가 있으면, 내장 함수 `range()` 가 편리합니다. 수열을 만듭니다:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

끝값은 만들어지는 수열에 포함되지 않습니다; `range(10)` 은 10개의 값을 만드는데, 길이 10인 시퀀스의 항목들을 가리키는 올바른 인덱스들입니다. 범위가 다른 숫자로 시작하거나, 다른 증가분을 (음수조차 가능합니다; 때로 이것을 ‘스텝(step)’ 이라고 부릅니다) 지정하는 것도 가능합니다:

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

시퀀스의 인덱스들로 이터레이트 하려면, 다음처럼 `range()` 와 `len()` 을 결합할 수 있습니다:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

하지만, 그럴 때 대부분은, `enumerate()` 함수를 쓰는 것이 편리합니다, 루프 테크닉 를 보세요.

범위를 그냥 인쇄하면 이상한 일이 일어납니다:

```
>>> range(10)
range(0, 10)
```

많은 경우에 `range()` 가 돌려준 객체는 리스트인 것처럼 동작하지만, 사실 리스트가 아닙니다. 이터레이터할 때 원하는 시퀀스 항목들을 순서대로 돌려주는 객체이지만, 실제로 리스트를 만들지 않아서 공간을 절약합니다.

이런 객체를 **이터러블** 이라고 부릅니다. 공급이 소진될 때까지 일련의 항목들을 얻을 수 있는 무엇인가를 기대하는 함수와 구조물들의 타깃으로 적합합니다. 우리는 `for` 문이 그런 구조물임을 보았습니다. 이터러블을 취하는 함수의 예는 `sum()` 입니다:

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [자료 구조](#), we will discuss in more detail about `list()`.

## 4.4 break and continue Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Found an odd number 7
Found an even number 8
Found an odd number 9
```

## 4.5 else Clauses on Loops

In a `for` or `while` loop the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred.

In a `while` loop, it's executed after the loop's condition becomes false.

In either kind of loop, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following `for` loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, **not** the `if` statement.)

One way to think of the `else` clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like `if/if/else`. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see [예외 처리하기](#).

## 4.6 pass 문

`pass` 문은 아무것도 하지 않습니다. 문법적으로 문장이 필요하지만, 프로그램이 특별히 할 일이 없을 때 사용할 수 있습니다. 예를 들어:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
... 
```

최소한의 클래스를 만들 때 흔히 사용됩니다:

```
>>> class MyEmptyClass:
...     pass
...
```

`pass`가 사용될 수 있는 다른 장소는 새 코드를 작업할 때 함수나 조건부 바디의 자리를 채우는 것인데, 여러분이 더 추상적인 수준에서 생각할 수 있게 합니다. `pass`는 조용히 무시됩니다:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
...
```

## 4.7 match Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a `switch` statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. `dataclasses`). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to ("x", "y"), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what. Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names (the `x=` and `y=` above) or class names (recognized by the "(...)" next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of `Points`, with `__match_args__` added, we could match it like this:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

We can add an `if` clause to a pattern, known as a "guard". If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")

```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the "bandwidth" and "latency" values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```

case (Point(x1, y1), Point(x2, y2) as p2): ...

```

will capture the second element of the input as `p2` (as long as the input is a sequence of two points)

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variable:

```

from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :)")

```

For a more detailed explanation and additional examples, you can look into [PEP 636](#) which is written in a tutorial format.

## 4.8 함수 정의하기

피보나치 수열을 임의의 한도까지 출력하는 함수를 만들 수 있습니다:

```

>>> def fib(n):    # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

...     a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

```

키워드 `def`는 함수 정의를 시작합니다. 함수 이름과 괄호로 싸인 형식 매개변수들의 목록이 뒤따릅니다. 함수의 바디를 형성하는 문장들이 다음 줄에서 시작되고, 반드시 들여쓰기 되어야 합니다.

함수 바디의 첫 번째 문장은 선택적으로 문자열 리터럴이 될 수 있습니다; 이 문자열 리터럴은 함수의 *도큐멘테이션 문자열*, 즉 *독스트링 (docstring)* 입니다. (독스트링에 대한 자세한 내용은 [도큐멘테이션 문자열](#) 에 나옵니다.) 독스트링을 사용해서 온라인이나 인쇄된 설명서를 자동 생성하거나, 사용자들이 대화형으로 코드를 열람할 수 있도록 하는 도구들이 있습니다; 여러분이 작성하는 코드에 독스트링을 첨부하는 것은 좋은 관습입니다, 그러니 버릇을 들이는 것이 좋습니다.

함수의 실행은 함수의 지역 변수들을 위한 새 심볼 테이블을 만듭니다. 좀 더 구체적으로, 함수에서의 모든 변수 대입들은 값을 지역 심볼 테이블에 저장합니다; 반면에 변수 참조는 먼저 지역 심볼 테이블을 본 다음, 전역 심볼 테이블을 본 후, 마지막으로 내장 이름들의 테이블을 살펴봅니다. 그래서, 참조될 수는 있다 하더라도, 전역 변수들과 둘러싸는 함수의 변수들은 함수 내에서 직접 값이 대입될 수 없습니다 (전역 변수를 `global` 문으로 명시하거나 둘러싸는 함수의 변수를 `nonlocal` 문으로 명시하지 않는 이상).

함수 호출로 전달되는 실제 매개변수들 (인자들)은 호출될 때 호출되는 함수의 지역 심볼 테이블에 만들어 집니다; 그래서 인자들은 값에 의한 호출 (*call by value*)로 전달됩니다 (값은 항상 객체의 값이 아니라 객체 참조입니다).<sup>1</sup> 함수가 다른 함수를 호출할 때, 또는 자신을 재귀적으로 호출할 때, 그 호출을 위한 새 지역 심볼 테이블이 만들어 집니다.

함수 정의는 함수 이름을 현재 심볼 테이블의 함수 객체와 연결합니다. 인터프리터는 해당 이름이 가리키는 객체를 사용자 정의 함수로 인식합니다. 다른 이름은 같은 함수 객체를 가리킬 수 있으며 함수에 액세스하는 데 사용될 수도 있습니다:

```

>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89

```

다른 언어들을 사용했다면, `fib` 가 값을 돌려주지 않기 때문에 함수가 아니라 프로시저라고 생각할 수 있습니다. 사실, `return` 문이 없는 함수도 값을 돌려줍니다, 비록 따분한 값이기는 하지만. 이 값은 `None` 이라고 불립니다 (내장 이름입니다). `None` 이 출력할 유일한 값이라면, 인터프리터는 보통 `None` 값 출력을 억제합니다. 꼭 보길 원한다면 `print()` 를 사용할 수 있습니다:

```

>>> fib(0)
>>> print(fib(0))
None

```

인쇄하는 대신, 피보나치 수열의 숫자들 리스트를 돌려주는 함수를 작성하는 것도 간단합니다:

```

>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result

```

(다음 페이지에 계속)

<sup>1</sup> 실제로, 객체 참조에 의한 호출 (*call by object reference*) 이 더 좋은 표현인데, 가변 객체가 전달되면, 호출자는 피호출자가 만든 변경을 볼 수 있기 때문입니다 (가령 리스트에 항목을 추가합니다).



(이전 페이지에서 계속)

```
...
>>> f100 = fib2(100)      # call it
>>> f100                  # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

여느 때처럼, 이 예는 몇 가지 새 파이썬 기능을 보여줍니다:

- `return` 문은 함수로부터 값을 갖고 복귀하게 만듭니다. 표현식 인자 없는 `return` 은 `None` 을 돌려줍니다. 함수의 끝으로 떨어지면 역시 `None` 을 돌려줍니다.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that ‘belongs’ to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see [클래스](#)) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

## 4.9 함수 정의 더 보기

정해지지 않은 개수의 인자들로 함수를 정의하는 것도 가능합니다. 세 가지 형식이 있는데, 조합할 수 있습니다.

### 4.9.1 기본 인자 값

가장 쓸모 있는 형식은 하나나 그 이상 인자들의 기본값을 지정하는 것입니다. 정의된 것보다 더 적은 개수의 인자들로 호출될 수 있는 함수를 만듭니다. 예를 들어:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
    print(reminder)
```

이 함수는 여러 가지 방법으로 호출될 수 있습니다:

- 오직 꼭 필요한 인자만 전달해서: `ask_ok('정말 끝 내길 원하세요?')`
- 선택적 인자 하나를 제공해서: `ask_ok('파일을 덮어써도 좋습니까?', 2)`
- 또는 모든 인자를 제공해서: `ask_ok('파일을 덮어써도 좋습니까?', 2, '자, 예나 아니요로만 답하세요!')`

이 예는 `in` 키워드도 소개하고 있습니다. 시퀀스가 어떤 값을 가졌는지 아닌지를 검사합니다.

기본값은 함수 정의 시점에 정의되고 있는 스코프에서 구해집니다, 그래서

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

는 5를 인쇄합니다.

**중요한 주의사항:** 기본값은 오직 한 번만 값이 구해집니다. 이것은 기본값이 리스트나 딕셔너리나 대부분 클래스의 인스턴스와 같은 가변 객체일 때 차이를 만듭니다. 예를 들어, 다음 함수는 계속되는 호출로 전달된 인자들을 누적합니다:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

다음과 같은 것을 인쇄합니다

```
[1]
[1, 2]
[1, 2, 3]
```

연속된 호출 간에 기본값이 공유되지 않기를 원한다면, 대신 함수를 이런 식으로 쓸 수 있습니다:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

## 4.9.2 키워드 인자

함수는 `kwarg=value` 형식의 키워드 인자를 사용해서 호출될 수 있습니다. 예를 들어, 다음 함수는:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

하나의 필수 인자(`voltage`)와 세 개의 선택적 인자(`state`, `action`, `type`)를 받아들입니다. 이 함수는 다음과 같은 방법 중 아무것으로나 호출될 수 있습니다.

```
parrot(1000)                                # 1 positional argument
parrot(voltage=1000)                        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')  # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)  # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

하지만 다음과 같은 호출들은 모두 올바르지 않습니다:

```
parrot()                                # required argument missing
parrot(voltage=5.0, 'dead')             # non-keyword argument after a keyword argument
parrot(110, voltage=220)                 # duplicate value for the same argument
parrot(actor='John Cleese')              # unknown keyword argument
```

함수 호출에서, 키워드 인자는 위치 인자 뒤에 나와야 합니다. 전달된 모든 키워드 인자는 함수가 받아들이는 인자 중 하나와 맞아야 하며 (예를 들어, `actor`는 `parrot` 함수의 올바른 인자가 아닙니다), 그 순서는 중요하지 않습니다. 이것들에는 필수 인자들도 포함됩니다 (예를 들어, `parrot(voltage=1000)`도 올바릅니다). 어떤 인자도 두 개 이상의 값을 받을 수 없습니다. 여기, 이 제약 때문에 실패하는 예가 있습니다:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

**\*\*name** 형식의 마지막 형식 매개변수가 존재하면, 형식 매개변수들에 대응하지 않는 모든 키워드 인자들을 담은 딕셔너리 (typesmapping 를 보세요) 를 받습니다. 이것은 **\*name** (다음 서브섹션에서 설명합니다) 형식의 형식 매개변수와 조합될 수 있는데, 형식 매개변수 목록 밖의 위치 인자들을 담은 튜플을 받습니다. (**\*name**은 **\*\*name** 앞에 나와야 합니다.) 예를 들어, 이런 함수를 정의하면:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("--" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

이런 식으로 호출될 수 있습니다:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

그리고 당연히 이렇게 인쇄합니다:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

인쇄되는 키워드 인자들의 순서 함수 호출로 전달된 순서와 일치함이 보장됨에 주목하세요.

### 4.9.3 특수 매개 변수

기본적으로, 인자는 위치나 명시적인 키워드로 파이썬 함수에 전달될 수 있습니다. 가독성과 성능을 위해, 개발자가 항목이 위치, 위치나 키워드 또는 키워드로 전달되는지를 판단할 때 함수 정의만을 보면 되도록, 인자가 전달될 방법을 제한하면 좋습니다.

함수 정의는 다음과 같습니다:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):  
    -----  
        |             |             |  
        |         Positional or keyword   |  
        |                                 - Keyword only  
    -- Positional only
```

여기서 /와 \*는 선택적입니다. 사용하면, 이 기호는 인자가 함수에 전달되는 방식에 따른 매개 변수의 종류를 나타냅니다: 위치 전용, 위치-키워드 및 키워드 전용. 키워드 매개 변수는 명명된 (named) 매개 변수라고도 합니다.

### 위치-키워드(Positional-or-Keyword) 인자

함수 정의에 /와 \*가 없으면, 인자를 위치나 키워드로 함수에 전달할 수 있습니다.

#### 위치 전용 매개 변수

좀 더 자세하게 살펴보면, 특정 매개 변수를 위치 전용으로 표시할 수 있습니다. 위치 전용이면, 매개 변수의 순서가 중요하며, 키워드로 매개 변수를 전달할 수 없습니다. 위치 전용 매개 변수는 / (슬래시) 앞에 놓입니다. /는 위치 전용 매개 변수를 나머지 매개 변수들로부터 논리적으로 분리하는 데 사용됩니다. 함수 정의에 /가 없으면, 위치 전용 매개 변수는 없습니다.

/ 다음의 매개 변수는 위치-키워드나 키워드 전용일 수 있습니다.

#### 키워드 전용 인자

매개 변수를 키워드 인자로 전달해야 함을 나타내도록, 매개 변수를 키워드 전용으로 표시하려면, 첫 번째 키워드 전용 매개 변수 바로 전에 인자 목록에 \*를 넣으십시오.

#### 함수 예제

/와 \* 마커에 주의를 기울이는 다음 예제 함수 정의를 고려하십시오:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

첫 번째 함수 정의 standard\_arg는 가장 익숙한 형식으로, 호출 규칙에 아무런 제한을 두지 않으며 인자는 위치나 키워드로 전달될 수 있습니다:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

두 번째 함수 pos\_only\_arg는 함수 정의에 /가 있으므로 위치 매개 변수만 사용하도록 제한됩니다:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword_
↪arguments: 'arg'
```

The third function kwd\_only\_arg only allows keyword arguments as indicated by a \* in the function definition:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

마지막은 같은 함수 정의에서 세 가지 호출 규칙을 모두 사용합니다:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword_
↪arguments: 'pos_only'
```

마지막으로, 위치 인자 `name`과 `name`을 키로 가지는 `**kwargs` 사이에 잠재적인 충돌이 있는 이 함수 정의를 고려하십시오:

```
def foo(name, **kwargs):
    return 'name' in kwargs
```

'name' 키워드는 항상 첫 번째 매개 변수에 결합하므로 True를 반환할 수 있는 호출은 불가능합니다. 예를 들면:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

그러나 /(위치 전용 인자)를 사용하면, `name`을 위치 인자로, 동시에 'name'을 키워드 인자의 키로 사용할 수 있으므로 가능합니다:

```
>>> def foo(name, /, **kwargs):
...     return 'name' in kwargs
...
>>> foo(1, **{'name': 2})
True
```

즉, 위치 전용 매개 변수의 이름을 `**kwargs`에서 모호함 없이 사용할 수 있습니다.

## 복습

사용 사례가 함수 정의에서 어떤 매개 변수를 사용할지 결정합니다:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

지침으로서:

- 매개 변수의 이름을 사용자가 사용할 수 없도록 하려면 위치 전용을 사용하십시오. 매개 변수 이름이 실제 의미가 없을 때, 함수가 호출될 때 인자의 순서를 강제하려고 할 때, 또는 일부 위치 매개 변수와 임의의 키워드를 받아들이고 싶을 때 유용합니다.
- 이름이 의미가 있고 함수 정의가 이름을 명시적으로 지정함으로써 더 이해하기 쉬워지거나, 사용자가 전달되는 인자의 위치에 의존하지 못하도록 하려면 키워드 전용을 사용하십시오.
- API의 경우, 향후 매개 변수의 이름이 수정될 때 비호환 API 변경이 발생하는 것을 방지하려면 위치 전용을 사용하십시오.

#### 4.9.4 임의의 인자 목록

마지막으로, 가장 덜 사용되는 옵션은 함수가 임의의 개수 인자로 호출될 수 있도록 지정하는 것입니다. 이 인자들은 튜플로 묶입니다 (튜플과 시퀀스 을 보세요). 가변 길이 인자 앞에, 없거나 여러 개의 일반 인자들이 올 수 있습니다.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normally, these *variadic* arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

#### 4.9.5 인자 목록 언패킹

인자들이 이미 리스트나 튜플에 있지만, 분리된 위치 인자들을 요구하는 함수 호출을 위해 언패킹 해야 하는 경우 반대 상황이 벌어집니다. 예를 들어, 내장 `range()` 함수는 별도의 *start*와 *stop* 인자를 기대합니다. 그것들이 따로 있지 않으면, 리스트와 튜플로부터 인자를 언패킹하기 위해 `*`-연산자를 사용해서 함수를 호출하면 됩니다:

```
>>> list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))         # call with arguments unpacked from a list
[3, 4, 5]
```

같은 방식으로 디렉터리도 `*`-연산자를 써서 키워드 인자를 전달할 수 있습니다:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin
↪ demised !
```

### 4.9.6 람다 표현식

lambda 키워드를 사용해서 작고 이름 없는 함수를 만들 수 있습니다. 이 함수는 두 인자의 합을 돌려줍니다: lambda a, b: a+b. 함수 객체가 있어야 하는 곳이면 어디나 람다 함수가 사용될 수 있습니다. 문법적으로는 하나의 표현식으로 제한됩니다. 의미적으로는, 일반적인 함수 정의의 편의 문법일 뿐입니다. 중첩된 함수 정의처럼, 람다 함수는 둘러싸는 스코프에 있는 변수들을 참조할 수 있습니다:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

위의 예는 함수를 돌려주기 위해 람다 표현식을 사용합니다. 또 다른 용도는 작은 함수를 인자로 전달하는 것입니다:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

### 4.9.7 도큐멘테이션 문자열

여기에 도큐멘테이션 문자열의 내용과 포매팅에 관한 몇 가지 관례가 있습니다.

첫 줄은 항상 객체의 목적을 짧고, 간결하게 요약해야 합니다. 간결함을 위해, 객체의 이름이나 형을 명시적으로 언급하지 않아야 하는데, 이것들은 다른 방법으로 제공되기 때문입니다 (이름이 함수의 작업을 설명하는 동사라면 예외입니다). 이 줄은 대문자로 시작하고 마침표로 끝나야 합니다.

도큐멘테이션 문자열에 여러 줄이 있다면, 두 번째 줄은 비어있어서, 시각적으로 요약과 나머지 설명을 분리해야 합니다. 뒤따르는 줄들은 하나나 그 이상의 문단으로, 객체의 호출 규약, 부작용 등을 설명해야 합니다.

파이썬 파서는 여러 줄 문자열 리터럴에서 들여쓰기를 제거하지 않기 때문에, 설명서를 처리하는 도구들은 필요하면 들여쓰기를 제거합니다. 이것은 다음과 같은 관례를 사용합니다. 문자열의 첫줄 뒤에 오는 첫 번째 비어있지 않은 줄이 전체 도큐멘테이션 문자열의 들여쓰기 수준을 결정합니다. (우리는 첫 줄을 사용할 수 없는데, 일반적으로 문자열을 시작하는 따옴표에 붙어있어서 들여쓰기가 문자열 리터럴의 것을 반영하지 않기 때문입니다.) 이 들여쓰기와 “동등한” 공백이 문자열의 모든 줄의 시작 부분에서 제거됩니다. 덜 들여쓰기 된 줄이 나타나지는 말아야 하지만, 나타난다면 모든 앞부분의 공백이 제거됩니다. 공백의 동등성은 탭 확장 (보통 8개의 스페이스) 후에 검사됩니다.

여기 여러 줄 독스트링의 예가 있습니다:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

## 4.9.8 함수 어노테이션

함수 어노테이션은 사용자 정의 함수가 사용하는 형들에 대한 완전히 선택적인 메타데이터 정보입니다 (자세한 내용은 [PEP 3107](#) 과 [PEP 484](#) 를 보세요).

*Annotations* are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

## 4.10 막간극: 코딩 스타일

이제 여러분은 파이썬의 더 길고, 더 복잡한 조각들을 작성하려고 합니다, 코딩 스타일에 대해 말할 적절한 시간입니다. 대부분 언어는 서로 다른 스타일로 작성될 (또는 더 간략하게, 포맷될) 수 있습니다; 어떤 것들은 다른 것들보다 더 읽기 쉽습니다. 다른 사람들이 여러분의 코드를 읽기 쉽게 만드는 것은 항상 좋은 생각이고, 훌륭한 코딩 스타일을 도입하는 것은 그렇게 하는 데 큰 도움을 줍니다.

파이썬을 위해, 대부분 프로젝트가 고수하는 스타일 가이드로 [PEP 8](#)이 나왔습니다; 이것은 매우 읽기 쉽고 눈이 편안한 코딩 스타일을 장려합니다. 모든 파이썬 개발자는 언젠가는 이 문서를 읽어야 합니다; 여러분을 위해 가장 중요한 부분들을 추려봤습니다:

- 들여 쓰기에 4-스페이스를 사용하고, 탭을 사용하지 마세요.  
4개의 스페이스는 작은 들여쓰기 (더 많은 중첩 도를 허락합니다) 와 큰 들여쓰기 (읽기 쉽습니다) 사이의 좋은 절충입니다. 탭은 혼란을 일으키고, 없애는 것이 최선입니다.
- 79자를 넘지 않도록 줄 넘김 하세요.  
이것은 작은 화면을 가진 사용자를 돕고 큰 화면에서는 여러 코드 파일들을 나란히 볼 수 있게 합니다.
- 함수, 클래스, 함수 내의 큰 코드 블록 사이에 빈 줄을 넣어 분리하세요.
- 가능하다면, 주석은 별도의 줄로 넣으세요.
- 독스트링을 사용하세요.
- 연산자들 주변과 콤마 뒤에 스페이스를 넣고, 괄호 바로 안쪽에는 스페이스를 넣지 마세요: `a = f(1, 2) + g(3, 4)`.
- 클래스와 함수들에 일관성 있는 이름을 붙이세요; 관례는 클래스의 경우 `UpperCamelCase`, 함수와 메서드의 경우 `lowercase_with_underscores`입니다. 첫 번째 메서드 인자의 이름으로는 항상 `self` 를 사용하세요 (클래스와 메서드에 대한 자세한 내용은 [클래스와의 첫 만남](#) 을 보세요).
- 여러분의 코드를 국제적인 환경에서 사용하려고 한다면 특별한 인코딩을 사용하지 마세요. 어떤 경우에도 파이썬의 기본, UTF-8, 또는 단순 ASCII조차, 이 최선입니다.
- 마찬가지로, 다른 언어를 사용하는 사람이 코드를 읽거나 유지할 약간의 가능성만 있더라도, 식별자에 ASCII 이외의 문자를 사용하지 마세요.



이 장에서는 여러분이 이미 배운 것들을 좀 더 자세히 설명하고, 몇 가지 새로운 것들을 덧붙입니다.

## 5.1 리스트 더 보기

리스트 자료 형은 몇 가지 메서드들을 더 갖고 있습니다. 이것들이 리스트 객체의 모든 메서드 들입니다:

`list.append(x)`

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

`list.extend(iterable)`

Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.

`list.insert(i, x)`

주어진 위치에 항목을 삽입합니다. 첫 번째 인자는 삽입되는 요소가 갖게 될 인덱스입니다. 그래서 `a.insert(0, x)` 는 리스트의 처음에 삽입하고, `a.insert(len(a), x)` 는 `a.append(x)` 와 동등합니다.

`list.remove(x)`

리스트에서 값이 `x` 와 같은 첫 번째 항목을 삭제합니다. 그런 항목이 없으면 `ValueError`를 일으킵니다.

`list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.

`list.clear()`

Remove all items from the list. Similar to `del a[:]`.

`list.index(x[, start[, end]])`

리스트에 있는 항목 중 값이 `x` 와 같은 첫 번째 것의 0부터 시작하는 인덱스를 돌려줍니다. 그런 항목이 없으면 `ValueError` 를 일으킵니다.

선택적인 인자 `start` 와 `end` 는 슬라이스 표기법처럼 해석되고, 검색을 리스트의 특별한 서브 시퀀스로 제한하는 데 사용됩니다. 돌려주는 인덱스는 `start` 인자가 아니라 전체 시퀀스의 시작을 기준으로 합니다.

`list.count(x)`

리스트에서 `x` 가 등장하는 횟수를 돌려줍니다.

`list.sort(* (Keyword-only parameters separator (PEP 3102)), key=None, reverse=False)`

리스트의 항목들을 제자리에서 정렬합니다 (인자들은 정렬 커스터마이제이션에 사용될 수 있습니다. 설명은 `sorted()` 를 보세요).

`list.reverse()`

리스트의 요소들을 제자리에서 뒤집습니다.

`list.copy()`

Return a shallow copy of the list. Similar to `a[:]`.

리스트 메서드 대부분을 사용하는 예:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4) # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`.<sup>1</sup> This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

### 5.1.1 리스트를 스택으로 사용하기

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
```

(다음 페이지에 계속)

<sup>1</sup> 다른 언어들에서는 가변 객체를 돌려주기도 하는데, `d->insert("a")->remove("b")->sort()`; 와 같은 메서드 연쇄를 허락합니다.

(이전 페이지에서 계속)

```
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

### 5.1.2 리스트를 큐로 사용하기

리스트를 큐로 사용하는 것도 가능한데, 처음으로 넣은 요소가 처음으로 꺼내지는 요소입니다 (“first-in, first-out”); 하지만, 리스트는 이 목적에는 효율적이지 않습니다. 리스트의 끝에 덧붙이거나, 끝에서 꺼내는 것은 빠르지만, 리스트의 머리에 덧붙이거나 머리에서 꺼내는 것은 느립니다 (다른 요소들을 모두 한 칸씩 이동시켜야 하기 때문입니다).

큐를 구현하려면, 양 끝에서의 덧붙이기와 꺼내기가 모두 빠르도록 설계된 `collections.deque` 를 사용하세요. 예를 들어:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()                # The first to arrive now leaves
'Eric'
>>> queue.popleft()                # The second to arrive now leaves
'John'
>>> queue                          # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

### 5.1.3 리스트 컴프리헨션

리스트 컴프리헨션은 리스트를 만드는 간결한 방법을 제공합니다. 흔한 용도는, 각 요소가 다른 시퀀스나 이터러블의 멤버들에 어떤 연산을 적용한 결과인 리스트를 만들거나, 어떤 조건을 만족하는 요소들로 구성된 서브 시퀀스를 만드는 것입니다.

예를 들어, 제곱수의 리스트를 만들고 싶다고 가정하자, 이런 식입니다:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

이것은 `x` 라는 이름의 변수를 만들고 (또는 덮어쓰고) 루프가 종료된 후에도 남아있게 만든다는 것에 유의하세요. 어떤 부작용도 없이, 제곱수의 리스트를 이런 식으로 계산할 수 있습니다:

```
squares = list(map(lambda x: x**2, range(10)))
```

또는, 이렇게 할 수도 있습니다:

```
squares = [x**2 for x in range(10)]
```

이것이 더 간결하고 읽기 쉽습니다.

리스트 컴프리헨션은 표현식과 그 뒤를 따르는 `for` 절과 없거나 여러 개의 `for` 나 `if` 절들을 감싸는 대괄호로 구성됩니다. 그 결과는 새 리스트인데, `for` 와 `if` 절의 문맥에서 표현식의 값을 구해서 만들어집니다. 예를 들어, 이 리스트 컴프리헨션은 두 리스트의 요소들을 서로 같지 않은 것끼리 결합합니다:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

그리고, 이것은 다음과 동등합니다:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

두 코드 조각에서 for 와 if 문의 순서가 같음에 유의하세요.

표현식이 튜플이면 (즉 앞의 예에서 (x, y)), 반드시 괄호로 둘러싸야 합니다.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1
    [x, x**2 for x in range(6)]
    ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

리스트 컴프리헨션은 복잡한 표현식과 중첩된 함수들을 포함할 수 있습니다:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

## 5.1.4 중첩된 리스트 컴프리헨션

리스트 컴프리헨션의 첫 표현식으로 임의의 표현식이 올 수 있는데, 다른 리스트 컴프리헨션도 가능합니다.

다음과 같은 길이가 4인 리스트 3개의 리스트로 구현된 3x4 행렬의 예를 봅시다:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

다음 리스트 컴프리헨션은 행과 열을 전치 시킵니다:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the inner list comprehension is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

이것은 다시 다음과 같습니다:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

실제 세상에서는, 복잡한 흐름문보다 내장 함수들을 선호해야 합니다. 이 경우에는 `zip()` 함수가 제 역할을 할 수 있습니다:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

이 줄에 나오는 에스터리스크에 대한 자세한 내용은 [인자 목록 언패킹](#) 을 보세요.

## 5.2 del 문

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` 는 변 자체를 삭제하는데에도 사용될 수 있습니다:

```
>>> del a
```

이후에 이름 `a` 를 참조하는 것은 에러입니다(적어도 다른 값이 새로 대입되기 전까지). 뒤에서 `del` 의 다른 용도를 보게 됩니다.

## 5.3 튜플과 시퀀스

리스트와 문자열이 인덱싱과 슬라이싱 연산과 같은 많은 성질을 공유함을 보았습니다. 이것들은 시퀀스 자료 형의 두 가지 예입니다 (`typeseq` 를 보세요). 파이썬은 진화하는 언어이기 때문에, 다른 시퀀스 자료 형이 추가될 수도 있습니다. 다른 표준 시퀀스 자료 형이 있습니다: 튜플 입니다.

튜플은 쉼표로 구분되는 여러 값으로 구성됩니다. 예를 들어:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

여러분이 보듯이, 출력되는 튜플은 항상 괄호로 둘러싸입니다, 그래서 중첩된 튜플이 올바르게 해석됩니다; 종종 괄호가 필요하기는 하지만 (튜플이 더 큰 표현식의 일부일 때), 둘러싼 괄호와 함께 또는 없이 입력될 수 있습니다. 튜플의 개별 항목에 대입하는 것은 가능하지 않지만, 리스트 같은 가변 객체를 포함하는 튜플을 만들 수는 있습니다.

튜플이 리스트처럼 보인다고 하더라도, 이것들은 다른 상황에서 다른 목적으로 사용됩니다. 튜플은 불변 이고, 보통 이질적인 요소들의 시퀀스를 포함합니다. 요소들은 언패킹 (이 섹션의 뒤에 나온다) 이나 인덱싱 (또는 네임드 튜플의 경우는 어트리뷰트로도) 으로 액세스합니다. 리스트는 가변 이고, 요소들은 보통 등질 적이고 리스트에 대한 이터레이션으로 액세스 됩니다.

특별한 문제는 비었거나 하나의 항목을 갖는 튜플을 만드는 것입니다: 이 경우를 수용하기 위해 문법은 추가적인 예외 사항을 갖고 있습니다. 빈 튜플은 빈 괄호 쌍으로 만들어집니다; 하나의 항목으로 구성된 튜플은 값 뒤에 쉼표를 붙여서 만듭니다 (값 하나를 괄호로 둘러싸기만 하는 것으로는 충분하지 않습니다). 추합니다, 하지만 효과적입니다. 예를 들어:

```
>>> empty = ()
>>> singleton = 'hello',      # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

문장 `t = 12345, 54321, 'hello!'` 는 튜플 패킹 의 예입니다: 값 `12345, 54321, 'hello!'` 는 함께 튜플 로 패킹 됩니다. 반대 연산 또한 가능합니다:

```
>>> x, y, z = t
```

이것은, 충분히 적절하게도, 시퀀스 언패킹 이라고 불리고 오른쪽에 어떤 시퀀스가 와도 됩니다. 시퀀스 언패킹은 등호의 좌변에 시퀀스에 있는 요소들과 같은 개수의 변수들이 올 것을 요구합니다. 다중 대입은 사실 튜플 패킹과 시퀀스 언패킹의 조합일뿐이라는 것에 유의하세요.

## 5.4 집합

파이썬은 집합을 위한 자료형도 포함합니다. 집합은 중복되는 요소가 없는 순서 없는 컬렉션입니다. 기본적인 용도는 멤버십 검사와 중복 엔트리 제거입니다. 집합 객체는 합집합, 교집합, 차집합, 대칭 차집합과 같은 수학적 연산들도 지원합니다.

집합을 만들 때는 중괄호나 `set()` 함수를 사용할 수 있습니다. 주의사항: 빈 집합을 만들려면 `set()` 을 사용해야 합니다. `{}` 가 아닙니다; 후자는 빈 딕셔너리를 만드는데, 다음 섹션에서 다룹니다.

여기 간략한 실연이 있습니다:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket           # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                            # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                        # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                        # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                        # letters in both a and b
{'a', 'c'}
>>> a ^ b                        # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

리스트 컴프리헨션과 유사하게, 집합 컴프리헨션도 지원됩니다:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

## 5.5 딕셔너리

Another useful data type built into Python is the *dictionary* (see typesmapping). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

딕셔너리를 (한 딕셔너리 안에서) 키가 중복되지 않는다는 제약조건을 가진 키: 값 쌍의 집합으로 생각하는 것이 최선입니다. 중괄호 쌍은 빈 딕셔너리를 만듭니다: `{}`. 중괄호 안에 쉼표로 분리된 키: 값 쌍들의 목록을 넣으면, 딕셔너리에 초기 키: 값 쌍들을 제공합니다; 이것이 딕셔너리가 출력되는 방식이기도 합니다.

딕셔너리의 주 연산은 값을 키와 함께 저장하고 주어진 키로 값을 추출하는 것입니다. `del` 로 키:값 쌍을 삭제하는 것도 가능합니다. 이미 사용하고 있는 키로 저장하면, 그 키로 저장된 예전 값은 잊힙니다. 존재하지 않는 키로 값을 추출하는 것은 에러입니다.

딕셔너리에 `list(d)` 를 수행하면 딕셔너리에서 사용되고 있는 모든 키의 리스트를 삽입 순서대로 돌려줍니다 (정렬을 원하면 대신 `sorted(d)` 를 사용하면 됩니다). 하나의 키가 딕셔너리에 있는지 검사하려면, `in` 키워드를 사용하세요.

여기에 딕셔너리를 사용하는 조그마한 예가 있습니다:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

`dict()` 생성자는 키-값 쌍들의 시퀀스로 부터 직접 딕셔너리를 구성합니다.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

이에 더해, 딕셔너리 컴프리헨션은 임의의 키와 값 표현식들로 부터 딕셔너리를 만드는데 사용될 수 있습니다:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

키가 간단한 문자열일 때, 때로 키워드 인자들을 사용해서 쌍을 지정하기가 쉽습니다:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

## 5.6 루프 테크닉

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

시퀀스를 루핑할 때, `enumerate()` 함수를 사용하면 위치 인덱스와 대응하는 값을 동시에 얻을 수 있습니다.



```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

둘이나 그 이상의 시퀀스를 동시에 루핑하려면, `zip()` 함수로 엔트리들의 쌍을 만들 수 있습니다.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

시퀀스를 거꾸로 루핑하려면, 먼저 정방향으로 시퀀스를 지정한 다음에 `reversed()` 함수를 호출하세요.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

정렬된 순서로 시퀀스를 루핑하려면, `sorted()` 함수를 사용해서 소스를 변경하지 않고도 정렬된 새 리스트를 받을 수 있습니다.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

시퀀스에 대해 `set()` 을 사용하면 중복 요소를 제거합니다. 시퀀스에 대해 `set()` 과 `sorted()` 를 함께 사용하는 것은 시퀀스의 고유한 요소를 정렬된 순서로 루핑하는 관용적 방법입니다.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

때로 루프를 돌고 있는 리스트를 변경하고픈 유혹을 느낍니다; 하지만, 종종, 대신 새 리스트를 만드는 것이 더 간단하고 더 안전합니다.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

## 5.7 조건 더 보기

while 과 if 문에서 사용되는 조건에는 비교뿐만 아니라 모든 연산자를 사용할 수 있습니다.

The comparison operators `in` and `not in` are membership tests that determine whether a value is in (or not in) a container. The operators `is` and `is not` compare whether two objects are really the same object. All comparison operators have the same priority, which is lower than that of all numerical operators.

비교는 연쇄할 수 있습니다. 예를 들어, `a < b == c` 는, `a` 가 `b` 보다 작고, 동시에 `b` 가 `c` 와 같은지 검사합니다.

비교는 논리 연산자 `and` 와 `or` 를 사용해서 결합할 수 있고, 비교의 결과는 (또는 그 밖의 모든 논리 표현식은) `not` 으로 부정될 수 있습니다. 이것들은 비교 연산자보다 낮은 우선순위를 갖습니다. 이것 간에는 `not` 이 가장 높은 우선순위를 갖고, `or` 가 가장 낮습니다. 그래서 `A and not B or C` 는 `(A and (not B)) or C` 와 동등합니다. 여느 때처럼, 원하는 조합을 표현하기 위해 괄호를 사용할 수 있습니다.

논리 연산자 `and` 와 `or` 는 소위 단락-회로(*short-circuit*) 연산자입니다: 인자들은 왼쪽에서 오른쪽으로 값이 구해지고, 결과가 결정되자마자 값 구하기는 중단됩니다. 예를 들어, `A` 와 `C` 가 참이고 `B` 가 거짓이면, `A and B and C` 는 표현식 `C` 의 값을 구하지 않습니다. 논리값이 아닌 일반 값으로 사용될 때, 단락-회로 연산자의 반환 값은 마지막으로 값이 구해진 인자입니다.

비교의 결과나 다른 논리 표현식의 결과를 변수에 대입할 수 있습니다. 예를 들어,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

파이썬에서, `C` 와는 달리, 표현식 안에서의 대입은 바다코끼리 연산자 `:=` 를 사용하여 명시적으로 수행해야 합니다. `C` 프로그램에서 흔히 마주치는 부류의 문제들을 회피하도록 합니다: `==` 를 사용할 표현식에 `=` 를 입력하는 실수.

## 5.8 시퀀스와 다른 형들을 비교하기

시퀀스 객체들은 보통 같은 시퀀스 형의 다른 객체들과 비교될 수 있습니다. 비교는 사전식 순서를 사용합니다: 먼저 첫 두 항목을 비교해서 다르면 이것이 비교의 결과를 결정합니다; 같으면, 다음 두 항목을 비교하고, 이런 식으로 어느 한 시퀀스가 소진될 때까지 계속합니다. 만약 비교되는 두 항목 자체가 같은 형의 시퀀스면, 사전식 비교가 재귀적으로 수행됩니다. 두 시퀀스의 모든 항목이 같다고 비교되면, 시퀀스들은 같은 것으로 취급됩니다. 한 시퀀스가 다른 하나의 머리 부분 서브 시퀀스면, 짧은 시퀀스가 작은 것입니다. 문자열의 사전식 배열은 개별 문자들의 순서를 정하는데 유니코드 코드 포인트 숫자를 사용합니다. 같은 형의 시퀀스들 간의 비교의 몇 가지 예는 이렇습니다:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

서로 다른 형의 객체들을 < 나 > 로 비교하는 것은, 그 객체들이 적절한 비교 메서드들을 갖고 있을 때만 허락된다는 것에 유의하세요. 예를 들어, 서로 다른 숫자 형들은 그들의 숫자 값에 따라 비교됩니다. 그래서 0은 0.0과 같고, 등등. 그렇지 않으면, 임의의 순서를 제공하는 대신, 인터프리터는 `TypeError` 를 일으킵니다.



파이썬 인터프리터를 종료한 후에 다시 들어가면, 여러분이 만들었던 정의들이 사라집니다 (함수나 변수들). 그래서, 좀 긴 프로그램을 쓰고자 한다면, 대신 인터프리터 입력을 편집기를 사용해서 준비한 후에 그 파일을 입력으로 사용해서 실행하는 것이 좋습니다. 이렇게 하는 것을 스크립트를 만든다고 합니다. 프로그램이 길어짐에 따라, 유지를 쉽게 하려고 여러 개의 파일로 나누고 싶을 수 있습니다. 여러 프로그램에서 썼던 편리한 함수를 각 프로그램에 정의를 복사하지 않고도 사용하고 싶을 수도 있습니다.

이런 것을 지원하기 위해, 파이썬은 정의들을 파일에 넣고 스크립트나 인터프리터의 대화형 모드에서 사용할 수 있는 방법을 제공합니다. 그런 파일을 모듈 이라고 부릅니다; 모듈로부터 정의들이 다른 모듈이나 메인 모듈로 임포트 될 수 있습니다 (메인 모듈은 최상위 수준에서 실행되는 스크립트나 계산기 모드에서 액세스하는 변수들의 컬렉션입니다).

모듈은 파이썬 정의와 문장들을 담고 있는 파일입니다. 파일의 이름은 모듈 이름에 확장자 `.py` 를 붙입니다. 모듈 내에서, 모듈의 이름은 전역 변수 `__name__` 으로 제공됩니다. 예를 들어, 여러분이 좋아하는 편집기로 `fibonacci.py` 라는 이름의 파일을 현재 디렉터리에 만들고 다음과 같은 내용으로 채웁니다:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

이제 파이썬 인터프리터에 들어가서 이 모듈을 다음과 같은 명령으로 임포트 합니다:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [파이썬 스코프와 이름 공간](#) for more details); it only adds the module name `fibo` there. Using the module name you can access

the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

함수를 자주 사용할 거라면 지역 이름으로 대입할 수 있습니다:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1 모듈 더 보기

모듈은 함수 정의뿐만 아니라 실행 가능한 문장들도 포함할 수 있습니다. 이 문장들은 모듈을 초기화하는데 사용됩니다. 이것들은 `import` 문에서 모듈 이름이 처음 등장할 때만 실행됩니다.<sup>1</sup> (이것들은 파일이 스크립트로 실행될 때도 실행됩니다.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

모듈이 정의하는 모든 이름을 임포트하는 변종도 있습니다:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

이것은 밑줄 (`_`)로 시작하는 것들을 제외한 모든 이름을 임포트 합니다. 대부분 파이썬 프로그래머들은 이 기능을 사용하지 않는데, 인터프리터로 알려지지 않은 이름들의 집합을 도입하게 되어, 여러분이 이미 정의한 것들을 가리게 될 수 있기 때문입니다.

일반적으로 모듈이나 패키지에서 `*`를 임포트하는 것은 눈살을 찌푸리게 한다는 것에 유의하세요, 종종 읽기에 편하지 않은 코드를 만들기 때문입니다. 하지만, 대화형 세션에서 입력을 줄이고자 사용하는 것은 상관없습니다.

모듈 이름 다음에 `as`가 올 경우, `as` 다음의 이름을 임포트한 모듈에 직접 연결합니다.

<sup>1</sup> In fact function definitions are also 'statements' that are 'executed'; the execution of a module-level function definition adds the function name to the module's global namespace.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

이것은 `import fibo` 가하는 것과 같은 방식으로 모듈을 임포트 하는데, 유일한 차이점은 그 모듈을 `fib` 라는 이름으로 사용할 수 있다는 것입니다.

`from` 을 써서 비슷한 효과를 낼 때도 사용할 수 있습니다:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

### 참고

효율성의 이유로, 각 모듈은 인터프리터 세션마다 한 번만 임포트됩니다. 그래서, 여러분이 모듈을 수정하면, 인터프리터를 다시 시작시켜야 합니다 — 또는, 대화형으로 시험하는 모듈이 하나뿐이라면, `importlib.reload()` 를 사용하세요. 예를 들어, `import importlib; importlib.reload(modulename)`.

## 6.1.1 모듈을 스크립트로 실행하기

여러분이 파이썬 모듈을 이렇게 실행하면

```
python fibo.py <arguments>
```

모듈에 있는 코드는, 그것을 임포트할 때처럼 실행됩니다. 하지만 `__name__` 은 `"__main__"` 로 설정됩니다. 이것은, 이 코드를 모듈의 끝에 붙여서:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

파일을 임포트할 수 있는 모듈뿐만 아니라 스크립트로도 사용할 수 있도록 만들 수 있음을 의미하는데, 오직 모듈이 “메인” 파일로 실행될 때만 명령행을 파싱하는 코드가 실행되기 때문입니다:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

모듈이 임포트될 때, 코드는 실행되지 않습니다:

```
>>> import fibo
>>>
```

이것은 종종 모듈에 대한 편리한 사용자 인터페이스를 제공하거나 테스트 목적으로 사용됩니다 (모듈을 스크립트로 실행하면 테스트 스위트 실행하기).

## 6.1.2 모듈 검색 경로

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- 입력 스크립트를 포함하는 디렉터리 (또는 파일이 지정되지 않았을 때는 현재 디렉터리).
- `PYTHONPATH` (디렉터리 이름들의 목록, 셸 변수 `PATH` 와 같은 문법).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at `sys-path-init`.

#### 참고

심볼릭 링크를 지원하는 파일 시스템에서, 입력 스크립트를 포함하는 디렉터리는 심볼릭 링크를 변환한 후에 계산됩니다. 다른 말로, 심볼릭 링크를 포함하는 디렉터리는 모듈 검색 경로에 포함되지 **않습니다**.

초기화 후에, 파이썬 프로그램은 `sys.path` 를 수정할 수 있습니다. 스크립트를 포함하는 디렉터리는 검색 경로의 처음에, 표준 라이브러리 경로의 앞에 놓입니다. 이것은 같은 이름일 경우 라이브러리 디렉터리에 있는 것 대신 스크립트를 포함하는 디렉터리의 것이 로드된다는 뜻입니다. 이 치환이 의도된 것이 아니라면 에러입니다. 더 자세한 정보는 [표준 모듈들](#) 을 보세요.

## 6.1.3 “컴파일된” 파이썬 파일

모듈 로딩을 빠르게 하려고, 파이썬은 `__pycache__` 디렉터리에 각 모듈의 컴파일된 버전을 `module.version.pyc` 라는 이름으로 캐싱합니다. `version` 은 컴파일된 파일의 형식을 지정합니다; 일반적으로 파이썬의 버전 번호를 포함합니다. 예를 들어, CPython 배포 3.3 에서 `spam.py` 의 컴파일된 버전은 `__pycache__/spam.cpython-33.pyc` 로 캐싱 됩니다. 이 명명법은 서로 다른 파이썬 배포와 버전의 컴파일된 모듈들이 공존할 수 있도록 합니다.

파이썬은 소스의 수정 시간을 컴파일된 버전과 비교해서 시효가 지나 다시 컴파일해야 하는지 검사합니다. 이것은 완전히 자동화된 과정입니다. 또한, 컴파일된 모듈은 플랫폼 독립적이기 때문에, 같은 라이브러리를 서로 다른 아키텍처를 갖는 시스템들에서 공유할 수 있습니다.

파이썬은 두 가지 상황에서 캐시를 검사하지 않습니다. 첫째로, 명령행에서 직접 로드되는 모듈들은 항상 재컴파일하고 그 결과를 저장하지 않습니다. 둘째로, 소스 모듈이 없으면 캐시를 검사하지 않습니다. 소스 없는 (컴파일된 파일만 있는) 배포를 지원하려면, 컴파일된 모듈이 소스 디렉터리에 있어야 하고, 소스 모듈이 없어야 합니다.

전문가를 위한 몇 가지 팁

- 컴파일된 모듈의 크기를 줄이려면 파이썬 명령에 `-O` 나 `-OO` 스위치를 사용할 수 있습니다. `-O` 스위치는 `assert` 문을 제거하고, `-OO` 스위치는 `assert` 문과 `__doc__` 문자열을 모두 제거합니다. 어떤 프로그램들은 이것들에 의존하기 때문에, 무엇을 하고 있는지 아는 경우만 이 옵션을 사용해야 합니다. “최적화된” 모듈은 `opt-` 태그를 갖고, 보통 더 작습니다. 미래의 배포에서는 최적화의 효과가 변경될 수 있습니다.
- `.py` 파일에서 읽을 때보다 `.pyc` 파일에서 읽을 때 프로그램이 더 빨리 실행되지는 않습니다; `.pyc` 파일에서 더 빨라지는 것은 로드되는 속도뿐입니다.
- 모듈 `compileall` 은 디렉터리에 있는 모든 모듈의 `.pyc` 파일들을 만들 수 있습니다.
- 이 절차에 대한 더 자세한 정보, 결정들의 순서도를 포함합니다, 는 [PEP 3147](#) 에 나옵니다.

## 6.2 표준 모듈들

파이썬은 표준 모듈들의 라이브러리가 함께 오는데, 별도의 문서 파이썬 라이브러리 레퍼런스 (이후로는 “라이브러리 레퍼런스”) 에서 설명합니다. 어떤 모듈들은 인터프리터에 내장됩니다; 이것들은 언어의 핵심적인 부분은 아니지만 그런데도 내장된 연산들에 대한 액세스를 제공하는데, 효율이나 시스템 호출과 같은 운영 체제 기본 요소들에 대한 액세스를 제공하기 위함입니다. 그런 모듈들의 집합은 설정 옵션인데 기반 플랫폼의 의존적입니다. 예를 들어, `winreg` 모듈은 윈도우 시스템에서만 제공됩니다. 특별한 모듈 하나는 주목을 받을 필요가 있습니다: `sys`. 모든 파이썬 인터프리터에 내장됩니다. 변수 `sys.ps1` 와 `sys.ps2` 는 기본과 보조 프롬프트로 사용되는 문자열을 정의합니다:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

이 두 개의 변수들은 인터프리터가 대화형 모드일 때만 정의됩니다.

변수 `sys.path` 는 인터프리터의 모듈 검색 경로를 결정하는 문자열들의 리스트입니다. 환경 변수 `PYTHONPATH` 에서 취한 기본 경로나, `PYTHONPATH` 가 설정되지 않는 경우 내장 기본값으로 초기화됩니다. 표준 리스트 연산을 사용해서 수정할 수 있습니다:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3 dir() 함수

내장 함수 `dir()` 은 모듈이 정의하는 이름들을 찾는 데 사용됩니다. 문자열들의 정렬된 리스트를 돌려줍니다:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fibo', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
'__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
'__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
'clear_type_cache', 'current_frames', 'debugmallocstats', 'framework',
'getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemcodeerrors', 'getfilesystemencoding', 'getprofile',
'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
'warnoptions']
```

인자가 없으면, `dir()` 는 현재 정의한 이름들을 나열합니다:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fibo', 'fib', 'sys']
```

모든 형의 이름을 나열한다는 것에 유의해야 합니다: 변수, 모듈, 함수, 등등.

`dir()` 은 내장 함수와 변수들의 이름을 나열하지 않습니다. 그것들의 목록을 원한다면, 표준 모듈 `builtins` 에 정의되어 있습니다:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
```

## 6.4 패키지

Packages are a way of structuring Python's module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

음향 파일과 과 음향 데이터의 일관된 처리를 위한 모듈들의 컬렉션 (“패키지”) 을 설계하길 원한다고 합시다. 여러 종류의 음향 파일 형식이 있으므로 (보통 확장자로 구분됩니다, 예를 들어: `.wav`, `.aiff`, `.au`), 다양한 파일 형식 간의 변환을 위해 계속 늘어나는 모듈들의 컬렉션을 만들고 유지할 필요가 있습니다. 또한, 음향 데이터에 적용하고자 하는 많은 종류의 연산들도 있으므로 (믹싱, 에코 넣기, 이퀄라이저 기능 적용, 인공적인 스테레오 효과 만들기와 같은), 이 연산들을 수행하기 위한 모듈들을 끊임없이 작성하게 될 것입니다. 패키지를 이렇게 구성해 볼 수 있습니다 (계층적 파일 시스템으로 표현했습니다):

<code>sound/</code>	Top-level package
<code>__init__.py</code>	Initialize the sound package
<code>formats/</code>	Subpackage for file format conversions
<code>__init__.py</code>	
<code>wavread.py</code>	
<code>wavwrite.py</code>	
<code>aiffread.py</code>	
<code>aiffwrite.py</code>	
<code>auread.py</code>	

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    auwrite.py
    ...
effects/                                Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
filters/                                Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

패키지를 임포트할 때, 파이썬은 `sys.path`에 있는 디렉터리들을 검색하면서 패키지 서브 디렉터리를 찾습니다.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a *namespace package*, a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

패키지 사용자는 패키지로부터 개별 모듈을 임포트할 수 있습니다, 예를 들어:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

서브 모듈을 임포트하는 다른 방법은 이렇습니다:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

또 다른 방법은 원하는 함수나 변수를 직접 임포트하는 것입니다:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

`from package import item`를 사용할 때, `item`은 패키지의 서브 모듈 (또는 서브 패키지)일 수도 있고 함수, 클래스, 변수 등 패키지에 정의된 다른 이름들일 수도 있음에 유의하세요. `import` 문은 먼저 `item`이 패키지에 정의되어 있는지 검사하고, 그렇지 않으면 모듈이라고 가정하고 로드를 시도합니다. 찾지 못한다면, `ImportError` 예외를 일으킵니다.

이에 반하여, `import item.subitem.subsubitem`와 같은 문법을 사용할 때, 마지막 것을 제외한 각 항목은 반드시 패키지여야 합니다; 마지막 항목은 모듈이나 패키지가 될 수 있지만, 앞의 항목에서 정의된 클래스, 함수, 변수 등이 될 수는 없습니다.

### 6.4.1 패키지에서 \* 임포트 하기

이제 `from sound.effects import *` 라고 쓰면 어떻게 될까? 이상적으로는, 어떻게든 파일 시스템에서 패키지에 어떤 모듈들이 들어있는지 찾은 다음, 그것들 모두를 임포트 하기를 원할 것입니다. 이렇게 하는 데는 시간이 오래 걸리고 서브 모듈을 임포트 함에 따라 어떤 서브 모듈을 명시적으로 임포트할 경우만 일어나야만 하는 원하지 않는 부수적 효과가 발생할 수 있습니다.

유일한 해결책은 패키지 저자가 패키지의 색인을 명시적으로 제공하는 것입니다. `import` 문은 다음과 같은 관계가 있습니다: 패키지의 `__init__.py` 코드가 `__all__` 이라는 이름의 목록을 제공하면, 이것을 `from package import *` 를 만날 때 임포트 해야만 하는 모듈 이름들의 목록으로 받아들입니다. 새 버전의 패키지를 출시할 때 이 목록을 최신 상태로 유지하는 것은 패키지 저자의 책임입니다. 패키지 저자가 패키지에서 `*` 를 임포트하는 용도가 없다고 판단한다면, 이것을 지원하지 않기로 할 수도 있습니다. 예를 들어, 파일 `sound/effects/__init__.py` 는 다음과 같은 코드를 포함할 수 있습니다:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound.effects` package.

Be aware that submodules might become shadowed by locally defined names. For example, if you added a `reverse` function to the `sound/effects/__init__.py` file, the `from sound.effects import *` would only import the two submodules `echo` and `surround`, but *not* the `reverse` submodule, because it is shadowed by the locally defined `reverse` function:

```
__all__ = [
    "echo",      # refers to the 'echo.py' file
    "surround",  # refers to the 'surround.py' file
    "reverse",   # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str): # <-- this name shadows the 'reverse.py' submodule
    return msg[::-1]   #         in the case of a 'from sound.effects import *'
```

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

설사 어떤 모듈이 `import *` 를 사용할 때 특정 패턴을 따르는 이름들만 익스포트 하도록 설계되었다 하더라도, 프로덕션 코드에서는 여전히 좋지 않은 사례로 여겨집니다.

`from package import specific_submodule` 을 사용하는데 잘못된 것은 없다는 것을 기억하세요! 사실, 임포트하는 모듈이 다른 패키지에서 같은 이름의 서브 모듈을 사용할 필요가 없는 한 권장되는 표기법입니다.

### 6.4.2 패키지 내부 간의 참조

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

상대 임포트가 현재 모듈의 이름에 기반을 둔다는 것에 주의하세요. 메인 모듈의 이름은 항상 `"__main__"` 이기 때문에, 파이썬 응용 프로그램의 메인 모듈로 사용될 목적의 모듈들은 반드시 절대 임포트를 사용해야 합니다.

### 6.4.3 여러 디렉터리에 있는 패키지

Packages support one more special attribute, `__path__`. This is initialized to be a *sequence* of strings containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

이 기능이 자주 필요하지는 않지만, 패키지에서 발견되는 모듈의 집합을 확장하는 데 사용됩니다.



프로그램의 출력을 표현하는 여러 가지 방법이 있습니다; 사람이 일기에 적합한 형태로 데이터를 인쇄할 수도 있고, 나중에 사용하기 위해 파일에 쓸 수도 있습니다. 이 장에서는 몇 가지 가능성을 논합니다.

## 7.1 장식적인 출력 포매팅

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

종종 단순히 스페이스로 구분된 값을 인쇄하는 것보다 출력 형식을 더 많이 제어해야 하는 경우가 있습니다. 출력을 포맷하는 데는 여러 가지 방법이 있습니다.

- 포맷 문자열 리터럴을 사용하려면, 시작 인용 부호 또는 삼중 인용 부호 앞에 `f` 또는 `F` 를 붙여 문자열을 시작하십시오. 이 문자열 안에서, `{` 및 `}` 문자 사이에, 변수 또는 리터럴 값을 참조할 수 있는 파이썬 표현식을 작성할 수 있습니다.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes  {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see `formatspec` for details).

- 마지막으로, 문자열 슬라이싱 및 이어붙이기 연산을 사용하여 상상할 수 있는 모든 배치를 만들으로써, 모든 문자열 처리를 스스로 수행할 수 있습니다. 문자열형에는 주어진 열 너비로 문자열을 채우는 데 유용한 연산을 수행하는 몇 가지 메서드가 있습니다.

장식적인 출력이 필요하지 않고 단지 디버깅을 위해 일부 변수를 빠르게 표시하려면, `repr()` 또는 `str()` 함수를 사용하여 모든 값을 문자열로 변환할 수 있습니다.

`str()` 함수는 어느 정도 사람이 읽기에 적합한 형태로 값의 표현을 돌려주게 되어 있습니다. 반면에 `repr()` 은 인터프리터에 의해 읽힐 수 있는 형태를 만들게 되어 있습니다 (또는 그렇게 표현할 수 있는 문법이 없으면 `SyntaxError` 를 일으키도록 구성됩니다). 사람이 소비하기 위한 특별한 표현이 없는 객체의 경우, `str()` 는 `repr()` 과 같은 값을 돌려줍니다. 많은 값, 숫자나 리스트와 딕셔너리와 같은 구조들, 은 두 함수를 쓸 때 같은 표현을 합니다. 특별히, 문자열은 두 가지 표현을 합니다.

몇 가지 예를 듭니다:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs'))
"'(32.5, 40000, ('spam', 'eggs'))"
```

`string` 모듈에는 문자열에 값을 치환하는 또 다른 방법을 제공하는 `Template` 클래스가 포함되어 있습니다. `$x`와 같은 자리 표시자를 사용하고 이것들을 딕셔너리에서 오는 값으로 치환하지만, 포매팅에 대한 제어를 훨씬 덜 제공합니다.

### 7.1.1 포맷 문자열 리터럴

포맷 문자열 리터럴(간단히 f-문자열이라고도 합니다)은 문자열에 `f` 또는 `F` 접두어를 붙이고 표현식을 `{expression}`로 작성하여 문자열에 파이썬 표현식의 값을 삽입할 수 있게 합니다.

선택적인 포맷 지정자가 표현식 뒤에 올 수 있습니다. 이것으로 값이 포맷되는 방식을 더 정교하게 제어할 수 있습니다. 다음 예는 원주율을 소수점 이하 세 자리로 반올림합니다.

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

`:'` 뒤에 정수를 전달하면 해당 필드의 최소 문자 폭이 됩니다. 열을 줄 맞춤할 때 편리합니다.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd      ==>      4127
Jack        ==>      4098
Dcab        ==>      7678
```

다른 수정자를 사용하면 포맷되기 전에 값을 변환할 수 있습니다. `!a`는 `ascii()` 를, `!s`는 `str()` 을, `!r`는 `repr()` 을 적용합니다.:



```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See self-documenting expressions for more information on the `=` specifier. For a reference on these format specifications, see the reference guide for the `formatspec`.

## 7.1.2 문자열 `format()` 메서드

`str.format()` 메서드의 기본적인 사용법은 이런 식입니다:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

중괄호와 그 안에 있는 문자들 (포맷 필드라고 부른다) 은 `str.format()` 메서드로 전달된 객체들로 치환됩니다. 중괄호 안의 숫자는 `str.format()` 메서드로 전달된 객체들의 위치를 가리키는데 사용될 수 있습니다.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

`str.format()` 메서드에 키워드 인자가 사용되면, 그 값들은 인자의 이름을 사용해서 지정할 수 있습니다.

```
>>> print('This {food} is {adjective}'.format(
...     food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

위치와 키워드 인자를 자유롭게 조합할 수 있습니다:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred',
...                                                other='Georg'))
The story of Bill, Manfred, and Georg.
```

나누고 싶지 않은 정말 긴 포맷 문자열이 있을 때, 포맷할 변수들을 위치 대신에 이름으로 지정할 수 있다면 좋을 것입니다. 간단히 덕셔너리를 넘기고 키를 액세스하는데 대괄호 `[]` 를 사용하면 됩니다.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k + '};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```

`str.format()` 를 사용한 문자열 포매팅의 완전한 개요는 `formatstrings` 을 보세요.

### 7.1.3 수동 문자열 포매팅

여기 같은 제곱수와 세제곱수 표를 수동으로 포매팅했습니다:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```

(`print()` 의 동작 방식으로 인해 각 칼럼 사이에 스페이스 하나가 추가되었음에 유의하세요: 항상 인자들 사이에 스페이스를 추가합니다.)

문자열 객체의 `str.rjust()` 메서드는 왼쪽에 스페이스를 채워서 주어진 폭으로 문자열을 우측 줄 맞추합니다. 비슷한 메서드 `str.ljust()` 와 `str.center()` 도 있습니다. 이 메서드들은 어떤 것도 출력하지 않습니다, 단지 새 문자열을 돌려줍니다. 입력 문자열이 너무 길면, 자르지 않고, 변경 없이 그냥 돌려줍니다; 이것이 열 배치를 엉망으로 만들겠지만, 보통 값에 대해 거짓말을 하게 될 대안보다는 낫습니다. (정말로 잘라내기를 원한다면, 항상 슬라이스 연산을 추가할 수 있습니다, `x.ljust(n)[:n]` 처럼.)

다른 메서드도 있습니다, `str.zfill()`. 숫자 문자열의 왼쪽에 0을 채웁니다. 플러스와 마이너스 부호도 이해합니다:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

### 7.1.4 예전의 문자열 포매팅

The `%` operator (modulo) can also be used for string formatting. Given *format* `%` values (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

더 자세한 내용은 `old-string-formatting` 섹션에 나옵니다.

## 7.2 파일을 읽고 쓰기

`open()` returns a *file object*, and is most commonly used with two positional arguments and one keyword argument: `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

첫 번째 인자는 파일 이름을 담은 문자열입니다. 두 번째 인자는 파일이 사용될 방식을 설명하는 몇 개의 문자들을 담은 또 하나의 문자열입니다. *mode* 는 파일을 읽기만 하면 `'r'`, 쓰기만 하면 `'w'` (같은 이름의 이미 존재하는 파일은 삭제됩니다) 가 되고, `'a'` 는 파일을 덧붙이기 위해 엽니다; 파일에 기록되는 모든 데이터는 자동으로 끝에 붙습니다. `'r+'` 는 파일을 읽고 쓰기 위해 엽니다. *mode* 인자는 선택적인데, 생략하면 `'r'` 이 가정됩니다.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific *encoding*. If *encoding* is not specified, the default is platform dependent (see `open()`). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you know that you need to use a different encoding. Appending a `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as *bytes* objects. You can not specify *encoding* when opening file in binary mode.

텍스트 모드에서, 읽을 때의 기본 동작은 플랫폼 의존적인 줄 종료 (유닉스에서 `\n`, 윈도우에서 `\r\n`) 를 단지 `\n` 로 변경하는 것입니다. 텍스트 모드로 쓸 때, 기본 동작은 `\n` 를 다시 플랫폼 의존적인 줄 종료로 변환하는 것입니다. 이 파일 데이터에 대한 무대 뒤의 수정은 텍스트 파일의 경우는 문제가 안 되지만, JPEG 이나 EXE 파일과 같은 바이너리 데이터를 망치게 됩니다. 그런 파일을 읽고 쓸 때 바이너리 모드를 사용하도록 주의하세요.

파일 객체를 다룰 때 `with` 키워드를 사용하는 것은 좋은 습관입니다. 예외는 도중 예외가 발생하더라도 스위치가 종료될 때 파일이 올바르게 닫힌다는 것입니다. `with` 를 사용하는 것은 동등한 `try-finally` 블록을 쓰는 것에 비교해 훨씬 짧기도 합니다:

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

`with` 키워드를 사용하지 않으면, `f.close()` 를 호출해서 파일을 닫고 사용된 시스템 자원을 즉시 반납해야 합니다.

**⚠ 경고**

`with` 키워드를 사용하거나 `f.close()` 를 호출하지 않고 `f.write()` 를 호출하면 프로그램이 성공적으로 종료되더라도 `f.write()` 의 인자가 디스크에 완전히 기록되지 않을 수 있습니다.

파일 객체가 닫힌 후에는, `with` 문이나 `f.close()` 를 호출하는 경우 모두, 파일 객체를 사용하려는 시도는 자동으로 실패합니다.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

## 7.2.1 파일 객체의 매소드

이 섹션의 나머지 예들은 `f` 라는 파일 객체가 이미 만들어졌다고 가정합니다.

파일의 내용을 읽으려면, `f.read(size)` 를 호출하는데, 일정량의 데이터를 읽고 문자열(텍스트 모드에서)이나 바이트열(바이너리 모드에서)로 돌려줍니다. `size` 는 선택적인 숫자 인자입니다. `size` 가 생략되거나 음수면 파일의 내용 전체를 읽어서 돌려줍니다; 파일의 크기가 기계의 메모리보다 두 배 크다면 여러분이 감당할 문제입니다. 그렇지 않으면 최대 `size` 문자(텍스트 모드에서)나 `size` 바이트(바이너리 모드에서)를 읽고 돌려줍니다. 파일의 끝에 도달하면, `f.read()` 는 빈 문자열('')을 돌려줍니다.

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` 은 파일에서 한 줄을 읽습니다; 개행 문자(`\n`)는 문자열의 끝에 보존되고, 파일이 개행문자로 끝나지 않는 때에만 파일의 마지막 줄에서만 생략됩니다. 이렇게 반환 값을 모호하지 않게 만듭니다; `f.readline()` 가 빈 문자열을 돌려주면, 파일의 끝에 도달한 것이지만, 빈 줄은 `'\n'`, 즉 하나의 개행문자만을 포함하는 문자열로 표현됩니다.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

파일에서 줄들을 읽으려면, 파일 객체에 대해 루핑할 수 있습니다. 이것은 메모리 효율적이고, 빠르며 간단한 코드로 이어집니다:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

파일의 모든 줄을 리스트로 읽어 들이려면 `list(f)` 나 `f.readlines()` 를 쓸 수 있습니다.

`f.write(string)` 은 `string` 의 내용을 파일에 쓰고, 출력된 문자들의 개수를 돌려줍니다.

```
>>> f.write('This is a test\n')
15
```

다른 형의 객체들은 쓰기 전에 변환될 필요가 있습니다 - 문자열(텍스트 모드에서)이나 바이트열 객체(바이너리 모드에서)로 -:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` 은 파일의 현재 위치를 가리키는 정수를 돌려주는데, 바이너리 모드의 경우 파일의 처음부터의 바이트 수로 표현되고 텍스트 모드의 경우는 불투명한 숫자입니다.

파일 객체의 위치를 바꾸려면, `f.seek(offset, whence)` 를 사용합니다. 위치는 기준점에 *offset* 을 더해져 계산됩니다; 기준점은 *whence* 인자로 선택합니다. *whence* 값이 0이면 파일의 처음부터 측정하고, 1이면 현재 파일 위치를 사용하고, 2 는 파일의 끝을 기준으로 사용합니다. *whence* 는 생략될 수 있고, 기본값은 0 이라서 파일의 처음을 기준으로 사용합니다.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5) # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

텍스트 파일에서는 (모드 문자열에 `b` 가 없이 열린 것들), 파일의 시작에 상대적인 위치 변경만 허락되고 (예외는 `seek(0, 2)` 를 사용해서 파일의 끝으로 위치를 변경하는 경우입니다), 올바른 *offset* 값은 `f.tell()` 이 돌려준 값과 0 뿐입니다. 그 밖의 다른 *offset* 값은 정의되지 않은 결과를 낳습니다.

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

## 7.2.2 json 으로 구조적인 데이터를 저장하기

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called **JSON (JavaScript Object Notation)**. The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

### 참고

JSON 형식은 데이터 교환을 위해 현대 응용 프로그램들이 자주 사용합니다. 많은 프로그래머가 이미 이것에 익숙하므로, 연동성을 위한 좋은 선택이 됩니다.

객체 `x` 가 있을 때, 간단한 한 줄의 코드로 그것의 JSON 문자열 표현을 볼 수 있습니다:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

`dump()` 라는 `dumps()` 함수의 변종은 객체를 텍스트 파일로 직렬화합니다. 그래서 `f` 가 쓰기를 위해 열린 텍스트 파일이면, 이렇게 할 수 있습니다:

```
json.dump(x, f)
```

To decode the object again, if `f` is a *binary file* or *text file* object which has been opened for reading:

```
x = json.load(f)
```

#### 참고

JSON files must be encoded in UTF-8. Use `encoding="utf-8"` when opening JSON file as a *text file* for both of reading and writing.

이 간단한 직렬화 테크닉이 리스트와 딕셔너리를 다룰 수 있지만, 임의의 클래스 인스턴스를 JSON 으로 직렬화하기 위해서는 약간의 수고가 더 필요합니다. `json` 모듈의 레퍼런스는 이 방법에 대한 설명을 담고 있습니다.

#### 더 보기

`pickle` - 피클 모듈

*JSON* 에 반해, *pickle* 은 임의의 복잡한 파이썬 객체들을 직렬화할 수 있는 프로토콜입니다. 파이썬에 국한되고 다른 언어로 작성된 응용 프로그램들과 통신하는데 사용될 수 없습니다. 기본적으로 안전하지 않기도 합니다: 믿을 수 없는 소스에서 온 데이터를 역 직렬화할 때, 숙련된 공격자에 의해 데이터가 조작되었다면 임의의 코드가 실행될 수 있습니다.

## 에러와 예외

지금까지 에러 메시지가 언급되지는 않았지만, 예제들을 직접 해보았다면 아마도 몇몇 개를 보았을 것입니다. (적어도) 두 가지 구별되는 에러들이 있습니다; 문법 에러 와 예외.

## 8.1 문법 에러

문법 에러는, 파싱 에러라고도 알려져 있습니다, 아마도 여러분이 파이썬을 배우고 있는 동안에는 가장 자주 만나는 종류의 불평일 것입니다:

```
>>> while True print('Hello world')
File "<stdin>", line 1
    while True print('Hello world')
                ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (':') is missing just before it.

The file name (<stdin> in our example) and line number are printed so you know where to look in case the input came from a file.

## 8.2 예외

문장이나 표현식이 문법적으로 올바르다 할지라도, 실행하려고 하면 에러를 일으킬 수 있습니다. 실행 중에 감지되는 에러들을 예외 라고 부르고 무조건 치명적이지는 않습니다: 파이썬 프로그램에서 이것들을 어떻게 다루는지 곧 배우게 됩니다. 하지만 대부분의 예외는 프로그램이 처리하지 않아서, 여기에서 볼 수 있듯이 에러 메시지를 만듭니다:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
        ~~~
ZeroDivisionError: division by zero
>>> 4 + spam*3
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
    ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~^~~
TypeError: can only concatenate str (not "int") to str
```

예러 메시지의 마지막 줄은 어떤 일이 일어났는지 알려줍니다. 예외는 여러 형으로 나타나고, 형이 메시지 일부로 인쇄됩니다: 이 예에서의 형은 `ZeroDivisionError`, `NameError`, `TypeError` 입니다. 예외 형으로 인쇄된 문자열은 발생한 내장 예외의 이름입니다. 이것은 모든 내장 예외들의 경우는 항상 참이지만, 사용자 정의 예외의 경우는 (편리한 관례임에도 불구하고) 꼭 그럴 필요는 없습니다. 표준 예외 이름은 내장 식별자입니다 (예약 키워드가 아닙니다).

줄의 나머지 부분은 예외의 형과 원인에 기반을 둔 상세 명세를 제공합니다.

예러 메시지의 앞부분은 스택 트레이스의 형태로 예외가 일어난 위치의 문맥을 보여줍니다. 일반적으로 소스의 줄들을 나열하는 스택 트레이스를 포함하고 있습니다; 하지만, 표준 입력에서 읽어 들인 줄들은 표시하지 않습니다.

`bltin-exceptions` 는 내장 예외들과 그들의 의미를 나열하고 있습니다.

## 8.3 예외 처리하기

선택한 예외를 처리하는 프로그램을 만드는 것이 가능합니다. 다음 예를 보면, 올바른 정수가 입력될 때까지 사용자에게 입력을 요청하지만, 사용자가 프로그램을 인터럽트 하는 것을 허용합니다 (`Control-C` 나 그 외에 운영 체제가 지원하는 것을 사용해서); 사용자가 만든 인터럽트는 `KeyboardInterrupt` 예외를 일으키는 형태로 나타남에 유의하세요.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops! That was no valid number. Try again...")
... 
```

`try` 문은 다음과 같이 동작합니다.

- 먼저, `try` 절 (`try` 와 `except` 사이의 문장들) 이 실행됩니다.
- 예외가 발생하지 않으면, `except` 절 을 건너뛰고 `try` 문의 실행은 종료됩니다.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the `try/except` block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:



```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except clause* may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     print(inst.args)    # arguments stored in .args
...     print(inst)         # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args    # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

The exception's `__str__()` output is printed as the last part ( 'detail' ) of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

The `try ... except` statement has an optional *else clause*, which, when present, must follow all *except clauses*. It is useful for code that must be executed if the *try clause* does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

`else` 절의 사용이 `try` 절에 코드를 추가하는 것보다 좋은데, `try ... except` 문에 의해 보호되고 있는 코드가 일으키지 않은 예외를 우연히 잡게 되는 것을 방지하기 때문입니다.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4 예외 일으키기

`raise` 문은 프로그래머가 지정한 예외가 발생하도록 강제할 수 있게 합니다. 예를 들어:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

만약 예외가 발생했는지는 알아야 하지만 처리하고 싶지는 않다면, 더 간단한 형태의 `raise` 문이 그 예외를 다시 일으킬 수 있게 합니다:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

## 8.5 예외 연쇄

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

이것은 예외를 변환할 때 유용할 수 있습니다. 예를 들면:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
    ~~~~^
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database

```

It also allows disabling automatic exception chaining using the `from None` idiom:

```

>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError from None
RuntimeError

```

연쇄 메커니즘에 대한 자세한 내용은, `bltin-exceptions`를 참조하십시오.

## 8.6 사용자 정의 예외

새 예외 클래스를 만듦으로써 프로그램은 자신의 예외에 이름을 붙일 수 있습니다 (파이썬 클래스에 대한 자세한 내용은 클래스를 보세요). 예외는 보통 직접적으로나 간접적으로 `Exception` 클래스를 계승합니다.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

대부분의 예외는 표준 예외들의 이름들과 유사하게, “Error”로 끝나는 이름으로 정의됩니다.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7 뒗정리 동작 정의하기

`try` 문은 또 다른 선택적 절을 가질 수 있는데 모든 상황에 실행되어야만 하는 뒗정리 동작을 정의하는 데 사용됩니다. 예를 들어:

```

>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt

```

finally 절이 있으면, try 문이 완료되기 전에 finally 절이 마지막 작업으로 실행됩니다. finally 절은 try 문이 예외를 생성하는지와 관계없이 실행됩니다. 다음은 예외가 발생할 때 더 복잡한 경우를 설명합니다:

- try 절을 실행하는 동안 예외가 발생하면, except 절에서 예외를 처리할 수 있습니다. 예외가 except 절에서 처리되지 않으면, finally 절이 실행된 후 예외가 다시 발생합니다.
- except 나 else 절 실행 중에 예외가 발생할 수 있습니다. 다시, finally 절이 실행된 후 예외가 다시 발생합니다.
- If the finally clause executes a break, continue or return statement, exceptions are not re-raised.
- try 문이 break, continue 또는 return 문에 도달하면, finally 절은 break, continue 또는 return 문 실행 직전에 실행됩니다.
- finally 절에 return 문이 포함되면, 반환 값은 try 절의 return 문이 주는 값이 아니라, finally 절의 return 문이 주는 값이 됩니다.

예를 들면:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

더 복잡한 예:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
    ~~~~~^~~~~~
  File "<stdin>", line 3, in divide
    result = x / y
            ~^^~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

보인 바와 같이, finally 절은 모든 경우에 실행됩니다. 두 문자열을 나눠서 발생한 TypeError 는 except 절에 의해 처리되지 않고 finally 절이 실행된 후에 다시 일어납니다.

실제 세상의 응용 프로그램에서, `finally` 절은 외부 자원을 사용할 때, 성공적인지 아닌지와 관계없이, 그 자원을 반납하는 데 유용합니다 (파일이나 네트워크 연결 같은 것들).

## 8.8 미리 정의된 뒷정리 동작들

어떤 객체들은 객체가 더 필요 없을 때 개입하는 표준 뒷정리 동작을 정의합니다. 그 객체를 사용하는 연산의 성공 여부와 관계없습니다. 파일을 열고 그 내용을 화면에 인쇄하려고 하는 다음 예를 보세요.

```
for line in open("myfile.txt"):
    print(line, end="")
```

이 코드의 문제점은 이 부분이 실행을 끝낸 뒤에도 예측할 수 없는 기간 동안 파일을 열린 채로 둔다는 것입니다. 간단한 스크립트에서는 문제가 되지 않지만, 큰 응용 프로그램에서는 문제가 될 수 있습니다. `with` 문은 파일과 같은 객체들이 즉시 올바르게 뒷정리 되도록 보장하는 방법을 제공합니다.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

문장이 실행된 후에, 줄을 처리하는 데 문제가 발생하더라도, 파일 `f` 는 항상 닫힙니다. 파일과 같이, 미리 정의된 뒷정리 동작들을 제공하는 객체들은 그들의 설명서에서 이 사실을 설명합니다.

## 8.9 Raising and Handling Multiple Unrelated Exceptions

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 3, in f
|       raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: {e}')
...
caught <class 'ExceptionGroup'>: e
>>>
```

By using `except*` instead of `except`, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each `except*` clause extracts from the group

exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be reraised.

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 2, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 2, in f
|       raise ExceptionGroup(
|         ...<12 lines>...
|     )
| ExceptionGroup: group1 (1 sub-exception)
+-+----- 1 -----
| ExceptionGroup: group2 (1 sub-exception)
+-+----- 1 -----
| RecursionError: 4
+-----
```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...     raise ExceptionGroup("Test Failures", excs)
...
>>>
```

## 8.10 Enriching Exceptions with Notes

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred.

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|       raise ExceptionGroup('We have some problems', excs)
| ExceptionGroup: We have some problems (3 sub-exceptions)
+----- 1 -----
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 2, in f
|       raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 1
+----- 2 -----
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 2, in f
|       raise OSError('operation failed')
| OSError: operation failed
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```
| Happened in Iteration 2
+----- 3 -----
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|     f()
|     ~^^
|   File "<stdin>", line 2, in f
|     raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 3
+-----
>>>
```



## 클래스

클래스는 데이터와 기능을 함께 묶는 방법을 제공합니다. 새 클래스를 만드는 것은 객체의 새 형을 만들어서, 그 형의 새 인스턴스를 만들 수 있도록 합니다. 각 클래스 인스턴스는 상태를 유지하기 위해 그 자신에게 첨부된 어트리뷰트를 가질 수 있습니다. 클래스 인스턴스는 상태를 바꾸기 위한 (클래스에 의해 정의된) 메서드도 가질 수 있습니다.

다른 프로그래밍 언어들과 비교할 때, 파이썬의 클래스 메커니즘은 최소한의 새로운 문법과 개념을 써서 클래스를 추가합니다. C++ 과 모듈라-3 에서 발견되는 클래스 메커니즘을 혼합합니다. 파이썬 클래스는 객체 지향형 프로그래밍의 모든 표준 기능들을 제공합니다: 클래스 상속 메커니즘은 다중 베이스 클래스를 허락하고, 자식 클래스는 베이스 클래스나 클래스들의 어떤 메서드도 재정의할 수 있으며, 메서드는 같은 이름의 베이스 클래스의 메서드를 호출할 수 있습니다. 객체들은 임의의 종류의 데이터를 양적 제한 없이 가질 수 있습니다. 모듈과 마찬가지로, 클래스는 파이썬의 동적인 본성을 함께 나눕니다: 실행 시간에 만들어지고, 만들어진 후에도 더 수정될 수 있습니다.

C++ 용어로, 보통 클래스 멤버들은 (데이터 멤버를 포함해서) *public* (예외는 아래 [비공개 변수](#)를 보세요) 하고, 모든 멤버 함수들은 *virtual* 입니다. 모듈라-3 처럼, 객체의 매소드에서 그 객체의 멤버를 참조하는 줄임 표현은 없습니다: 메서드 함수는 그 객체를 표현하는 명시적인 첫 번째 인자를 선언하는데, 함수 호출 때 묵시적으로 제공됩니다. 스톱토크처럼, 클래스 자신도 객체입니다. 이것이 임포트링과 이름 변경을 위한 개념을 제공합니다. C++ 나 모듈라-3 와는 달리, 내장형도 사용자가 확장하기 위해 베이스 클래스로 사용할 수 있습니다. 또한, C++ 처럼, 특별한 문법을 갖는 대부분의 내장 연산자들은 (산술 연산자, 서브스크립팅, 등등) 클래스 인스턴스에 대해 새로 정의될 수 있습니다.

(클래스에 대해 보편적으로 받아들여지는 용어들이 없는 상태에서, 이따금 스톱토크나 C++ 용어들을 사용할 것입니다. C++ 보다 객체 지향적 개념들이 파이썬의 것과 더 가까우므로 모듈라-3 용어를 사용할 수도 있지만, 들어본 독자들이 별로 없을 것으로 예상합니다.)

## 9.1 이름과 객체에 관한 한마디

객체는 개체성(individuality)을 갖고, 여러 개의 이름이 (여러 개의 스코프에서) 같은 객체에 연결될 수 있습니다. 이것은 다른 언어들에서는 에일리어싱(aliasing) 이라고 알려져 있습니다. 보통 파이썬을 처음 볼 때 이 점을 높이 평가하지는 않고, 불변 기본형들 (숫자, 문자열, 튜플)을 다루는 동안은 안전하게 무시할 수 있습니다. 하지만, 에일리어싱은 리스트, 딕셔너리나 그 밖의 다른 가변 객체들을 수반하는 파이썬 코드의 의미에 극적인 효과를 줄 수 있습니다. 이것은 보통 프로그램에 혜택이 되는데, 에일리어스는 어떤 면에서 포인터처럼 동작하기 때문입니다. 예를 들어, 구현이 포인터만 전달하기 때문에, 객체를 전달하는 비용이 적게 듭니다; 그리고 함수가 인자로 전달된 객체를 수정하면, 호출자는 그 변경을 보게 됩니다 — 이것은 파스칼에서 사용되는 두 가지 서로 다른 인자 전달 메커니즘의 필요를 제거합니다.

## 9.2 파이썬 스코프와 이름 공간

클래스를 소개하기 전에, 파이썬의 스코프 규칙에 대해 몇 가지 말할 것이 있습니다. 클래스 정의는 이름 공간으로 깔끔한 요령을 부리고, 여러분은 무엇이 일어나는지 완전히 이해하기 위해 스코프와 이름 공간이 어떻게 동작하는지 알 필요가 있습니다. 덧붙여 말하자면, 이 주제에 대한 지식은 모든 고급 파이썬 프로그래머에게 쓸모가 있습니다.

몇 가지 정의로 시작합시다.

이름 공간은 이름에서 객체로 가는 매핑입니다. 대부분의 이름 공간은 현재 파이썬 디렉터리로 구현되어 있지만, 보통 다른 식으로는 알아차릴 수 없고 (성능은 예외입니다), 앞으로는 바뀔 수 있습니다. 이름 공간의 예는: 내장 이름들의 집합 (`abs()`와 같은 함수들과 내장 예외 이름들을 포함합니다); 모듈의 전역 이름들; 함수 호출에서의 지역 이름들. 어떤 의미에서 객체의 어트리뷰트 집합도 이름 공간을 형성합니다. 이름 공간에 대해 알아야 할 중요한 것은 서로 다른 이름 공간들의 이름 간에는 아무런 관계가 없다는 것입니다; 예를 들어, 두 개의 서로 다른 모듈들은 모두 혼동 없이 함수 `maximize`를 정의할 수 있습니다 — 모듈의 사용자들은 모듈 이름을 앞에 붙여야 합니다.

그런데, 저는 어트리뷰트라는 단어를 점 뒤에 오는 모든 이름에 사용합니다 — 예를 들어, 표현식 `z.real`에서, `real`는 객체 `z`의 어트리뷰트입니다. 엄밀하게 말해서, 모듈에 있는 이름들에 대한 참조는 어트리뷰트 참조입니다: 표현식 `modname.funcname`에서, `modname`은 모듈 객체고 `funcname`는 그것의 어트리뷰트입니다. 이 경우에는 우연히도 모듈의 어트리뷰트와 모듈에서 정의된 전역 이름 간에 직접적인 매핑이 생깁니다: 같은 이름 공간을 공유합니다!

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

이름 공간들은 서로 다른 순간에 만들어지고 서로 다른 수명을 갖습니다. 내장 이름들을 담는 이름 공간은 파이썬 인터프리터가 시작할 때 만들어지고 영원히 지워지지 않습니다. 모듈의 전역 이름 공간은 모듈 정의를 읽는 동안 만들어집니다; 보통, 모듈 이름 공간은 인터프리터가 끝날 때까지 남습니다. 인터프리터의 최상위 호출 때문에 실행되는, 스크립트 파일이나 대화형으로 읽히는, 문장들은 `__main__`이라고 불리는 모듈 일부로 여겨져서 그들 자신의 이름 공간을 갖습니다. (내장 이름들 또한 모듈에 속하는데; 이것을 `builtins`라 부릅니다.)

함수의 지역 이름 공간은 함수가 호출될 때 만들어지고, 함수가 복귀하거나 함수 내에서 처리되지 않는 예외를 일으킬 때 삭제됩니다. (사실, 잊어버린다는 것이 실제로 일어나는 일에 대한 더 좋은 설명입니다.) 물론, 재귀적 호출은 각각 자기 자신만의 지역 이름 공간을 갖습니다.

스코프는 이름 공간을 직접 액세스할 수 있는 파이썬 프로그램의 텍스트적인 영역입니다. 여기에서 “직접 액세스 가능한”이란 이름에 대한 정규화되지 않은 참조가 그 이름 공간에서 이름을 찾으려고 시도한다는 의미입니다.

스코프가 정적으로 결정됨에도 불구하고, 동적으로 사용됩니다. 실행 중 어느 시점에서건, 이름 공간을 직접 액세스 가능한, 세 개나 네 개의 중첩된 스코프가 있습니다:

- 가장 먼저 검색되는, 가장 내부의 스코프는 지역 이름들을 포함합니다
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- 마지막 직전의 스코프는 현재 모듈의 전역 이름들을 포함합니다
- (가장 나중에 검색되는) 가장 외부의 스코프는 내장 이름들을 포함하고 있는 이름 공간입니다.

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

<sup>1</sup> Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

보통, 지역 스코프는 현재 함수의 지역 이름들을 (텍스트 적으로) 참조합니다. 함수 바깥에서, 지역 스코프는 전역 스코프와 같은 이름 공간을 참조합니다: 모듈의 이름 공간. 클래스 정의들은 지역 스코프에 또 하나의 이름 공간을 배치합니다.

스코프가 텍스트 적으로 결정된다는 것을 깨닫는 것은 중요합니다: 모듈에서 정의된 함수의 전역 스코프는, 어디에서 어떤 에일리어스를 통해 그 함수가 호출되는지에 관계없이, 그 모듈의 이름 공간입니다. 반면에, 이름을 실제로 검색하는 것은 실행시간에 동적으로 수행됩니다 — 하지만, 언어 정의는 컴파일 시점의 정적인 이름 결정을 향해 진화하고 있어서, 동적인 이름 결정에 의존하지 말아야 합니다! (사실, 지역 변수들은 이미 정적으로 결정됩니다.)

파이썬의 특별한 특징은 - `global` 이나 `nonlocal` 문이 없을 때 - 이름에 대입하면 항상 가장 내부의 스코프로 간다는 것입니다. 대입은 데이터를 복사하지 않습니다 - 이름을 단지 객체에 연결할 뿐입니다. 삭제도 마찬가지입니다: 문장 `del x` 는 지역 스코프가 참조하는 이름 공간에서 `x` 의 연결을 제거합니다. 사실, 새 이름을 소개하는 모든 연산은 지역 스코프를 사용합니다: 특히, `import` 문과 함수 정의는 모듈이나 함수 이름을 지역 스코프에 연결합니다.

`global` 문은 특정 변수가 전역 스코프에 있으며 그곳에 재연결되어야 함을 가리킬 때 사용될 수 있습니다; `nonlocal` 문은 특정 변수가 둘러싸는 스코프에 있으며 그곳에 재연결되어야 함을 가리킵니다.

## 9.2.1 스코프와 이름 공간 예

이것은 어떻게 서로 다른 스코프와 이름 공간을 참조하고, `global` 과 `nonlocal` 이 변수 연결에 어떤 영향을 주는지를 보여주는 예입니다:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

예제 코드의 출력은 이렇게 됩니다:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

어떻게 지역 대입이 (이것이 기본입니다) `scope_test` 의 `spam` 연결을 바꾸지 않는지에 유의하세요. `nonlocal` 대입은 `scope_test` 의 `spam` 연결을 바꾸고 `global` 대입은 모듈 수준의 연결을 바꿉니다.

`global` 대입 전에는 `spam` 의 연결이 없다는 것도 볼 수 있습니다.

## 9.3 클래스와의 첫 만남

클래스는 약간의 새 문법과 세 개의 객체형과 몇 가지 새 개념들을 도입합니다.

### 9.3.1 클래스 정의 문법

클래스 정의의 가장 간단한 형태는 이렇게 생겼습니다:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

함수 정의(`def` 문)처럼, 클래스 정의는 어떤 효과가 생기기 위해서는 먼저 실행되어야 합니다. (상상컨대 클래스 정의를 `if` 문의 분기나 함수 내부에 놓을 수 있습니다)

실재적으로, 클래스 정의 내부의 문장들은 보통 함수 정의들이지만, 다른 문장들도 허락되고 때로 쓸모가 있습니다 — 나중에 이 주제로 돌아올 것입니다. 클래스 내부의 함수 정의는 보통, 메서드 호출 규약의 영향을 받은, 특별한 형태의 인자 목록을 갖습니다. — 다시, 이것은 뒤에서 설명됩니다.

클래스 정의에 진입할 때, 새 이름 공간이 만들어지고 지역 스코프로 사용됩니다 — 그래서, 모든 지역 변수들의 대입은 이 새 이름 공간으로 갑니다. 특히, 함수 정의는 새 함수의 이름을 이곳에 연결합니다.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

### 9.3.2 클래스 객체

클래스 객체는 두 종류의 연산을 지원합니다: 어트리뷰트 참조와 인스턴스 만들기.

어트리뷰트 참조는 파이썬의 모든 어트리뷰트 참조에 사용되는 표준 문법을 사용합니다: `obj.name`. 올바른 어트리뷰트 이름은 클래스 객체가 만들어질 때 클래스의 이름 공간에 있던 모든 이름입니다. 그래서, 클래스 정의가 이렇게 될 때:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

클래스 인스턴스 만들기는 함수 표기법을 사용합니다. 클래스 객체가 클래스의 새 인스턴스를 돌려주는 매개변수 없는 함수인 체합니다. 예를 들어 (위의 클래스를 가정하면):

```
x = MyClass()
```

는 클래스의 새 인스턴스를 만들고 이 객체를 지역 변수 `x` 에 대입합니다.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

### 9.3.3 인스턴스 객체

이제 인스턴스 객체로 무엇을 할 수 있을까? 인스턴스 객체가 이해하는 오직 한가지 연산은 어트리뷰트 참조입니다. 두 가지 종류의 올바른 어트리뷰트 이름이 있습니다: 데이터 어트리뷰트와 메서드.

*data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object.

인스턴스 객체의 올바른 메서드 이름은 그것의 클래스에 달려있습니다. 정의상, 함수 객체인 클래스의 모든 어트리뷰트들은 상응하는 인스턴스의 메서드들을 정의합니다. 그래서 우리의 예제에서, `x.f` 는 올바른 메서드 참조인데, `MyClass.f` 가 함수이기 때문입니다. 하지만 `x.i` 는 그렇지 않은데, `MyClass.i` 가 함수가 아니기 때문입니다. 그러나, `x.f` 는 `MyClass.f` 와 같은 것이 아닙니다 — 이것은 함수 객체가 아니라 메서드 객체입니다.

### 9.3.4 메서드 객체

보통, 메서드는 연결되자마자 호출됩니다:

```
x.f()
```

In the `MyClass` example, this will return the string 'hello world'. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

는 영원히 계속 `hello world` 를 인쇄합니다.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

실제로, 여러분은 답을 짐작할 수 있습니다: 메서드의 특별함은 인스턴스 객체가 함수의 첫 번째 인자로 전달된다는 것입니다. 우리 예에서, 호출 `x.f()` 는 정확히 `MyClass.f(x)` 와 동등합니다. 일반적으로,  $n$  개의 인자들의 목록으로 메서드를 호출하는 것은, 첫 번째 인자 앞에 메서드의 인스턴스 객체를 삽입해서 만든 인자 목록으로 상응하는 함수를 호출하는 것과 동등합니다.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

### 9.3.5 클래스와 인스턴스 변수

일반적으로 말해서, 인스턴스 변수는 인스턴스별 데이터를 위한 것이고 클래스 변수는 그 클래스의 모든 인스턴스에서 공유되는 어트리뷰트와 메서드를 위한 것입니다:

```
class Dog:

    kind = 'canine'           # class variable shared by all instances

    def __init__(self, name):
        self.name = name     # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # shared by all dogs
'canine'
>>> e.kind                # shared by all dogs
'canine'
>>> d.name                # unique to d
'Fido'
>>> e.name                # unique to e
'Buddy'
```

이름과 객체에 관한 한마디 에서 논의했듯이, 리스트나 딕셔너리와 같은 가변 객체가 참여할 때 공유 데이터는 예상치 못한 효과를 줄 가능성이 있습니다. 예를 들어, 다음 코드에서 `tricks` 리스트는 클래스 변수로 사용되지 않아야 하는데, 하나의 리스트가 모든 `Dog` 인스턴스들에 공유되기 때문입니다.

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

대신, 클래스의 올바른 설계는 인스턴스 변수를 사용해야 합니다:

```
class Dog:
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```

def __init__(self, name):
    self.name = name
    self.tricks = []    # creates a new empty list for each dog

def add_trick(self, trick):
    self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']

```

## 9.4 기타 주의사항들

인스턴스와 클래스 모두에서 같은 어트리뷰트 이름이 등장하면, 어트리뷰트 조회는 인스턴스를 우선합니다:

```

>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east

```

데이터 어트리뷰트는 메서드 뿐만 아니라 객체의 일반적인 사용자 (“클라이언트”)에 의해서 참조될 수도 있습니다. 달리 표현하면, 클래스는 순수하게 추상적인 데이터형을 구현하는데 사용될 수 없습니다. 사실, 파이썬에서는 데이터 은닉을 강제할 방법이 없습니다 — 모두 관례에 의존합니다. (반면에, C로 작성된 파이썬 구현은 필요하다면 구현 상세를 완전히 숨기고 객체에 대한 액세스를 제어할 수 있습니다; 이것은 C로 작성된 파이썬 확장에서 사용될 수 있습니다.)

클라이언트는 데이터 어트리뷰트를 조심스럽게 사용해야 합니다 — 클라이언트는 데이터 어트리뷰트를 건드려서 메서드들에 의해 유지되는 불변성들을 망가뜨릴 수 있습니다. 클라이언트는 이름 충돌을 피하는 한 메서드들의 유효성을 손상하지 않고도 그들 자신의 데이터 어트리뷰트를 인스턴스 객체에 추가할 수도 있음에 유의하세요 — 다시 한번, 명명 규칙은 여러 골칫거리를 피할 수 있게 합니다.

메서드 안에서 데이터 어트리뷰트들(또는 다른 메서드들!)을 참조하는 줄임 표현은 없습니다. 저는 이것이 실제로 메서드의 가독성을 높인다는 것을 알게 되었습니다: 메서드를 훑어볼 때 지역 변수와 인스턴스 변수를 혼동할 우려가 없습니다.

종종, 메서드의 첫 번째 인자는 `self` 라고 불립니다. 이것은 관례일 뿐입니다: 이름 `self` 는 파이썬에서 아무런 특별한 의미를 갖지 않습니다. 하지만, 이 규칙을 따르지 않을 때 여러분의 코드가 다른 파이썬 프로그래머들이 읽기에 불편하고, 클래스 브라우저 프로그램도 이런 규칙에 의존하도록 작성되었다고 상상할 수 있음에 유의하세요.

클래스 어트리뷰트인 모든 함수는 그 클래스의 인스턴스들을 위한 메서드를 정의합니다. 함수 정의가 클래스 정의에 텍스트 적으로 둘러싸일 필요는 없습니다: 함수 객체를 클래스의 지역 변수로 대입하는 것 역시 가능합니다. 예를 들어:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

메서드는 `self` 인자의 메서드 어트리뷰트를 사용해서 다른 메서드를 호출할 수 있습니다:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

메서드는 일반 함수들과 마찬가지로 전역 이름을 참조할 수 있습니다. 메서드에 결합한 전역 스코프는 그것의 정의를 포함하는 모듈입니다. (클래스는 결코 전역 스코프로 사용되지 않습니다.) 메서드에서 전역 데이터를 사용할 좋은 이유를 거의 만나지 못하지만, 전역 스코프를 정당하게 사용하는 여러 가지 경우가 있습니다: 한가지는, 전역 스코프에 정의된 함수와 메서드 뿐만 아니라, 그곳에 임포트된 함수와 모듈도 메서드가 사용할 수 있다는 것입니다. 보통, 메서드를 포함하는 클래스 자신은 이 전역 스코프에 정의되고, 다음 섹션에서 메서드가 자신의 클래스를 참조하길 원하는 몇 가지 좋은 이유를 보게 될 것입니다.

각 값은 객체고, 그러므로 클래스 (형 이라고도 불린다) 를 갖습니다. 이것은 `object.__class__` 에 저장되어 있습니다.

## 9.5 상속

물론, 상속을 지원하지 않는다면 언어 기능은 “클래스”라는 이름을 불일만한 가치가 없을 것입니다. 파생 클래스 정의의 문법은 이렇게 생겼습니다:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a namespace accessible from the scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

파생 클래스 정의의 실행은 베이스 클래스와 같은 방식으로 진행됩니다. 클래스 객체가 만들어질 때,

베이스 클래스가 기억됩니다. 이것은 어트리뷰트 참조를 결정할 때 사용됩니다: 요청된 어트리뷰트가 클래스에서 발견되지 않으면 베이스 클래스로 검색을 확장합니다. 베이스 클래스 또한 다른 클래스로부터 파생되었다면 이 규칙은 재귀적으로 적용됩니다.

파생 클래스의 인스턴스 만들기에 특별한 것은 없습니다: `DerivedClassName()` 는 그 클래스의 새 인스턴스를 만듭니다. 메서드 참조는 다음과 같이 결정됩니다: 대응하는 클래스 어트리뷰트가 검색되는데, 필요하면 베이스 클래스의 연쇄를 타고 내려갑니다. 이것이 함수 객체를 준다면 메서드 참조는 올바릅니다.

파생 클래스는 베이스 클래스의 메서드들을 재정의할 수 있습니다. 메서드가 같은 객체의 다른 메서드를 호출할 때 특별한 권한 같은 것은 없으므로, 베이스 클래스에 정의된 다른 메서드를 호출하는 베이스 클래스의 메서드는 재정의된 파생 클래스의 메서드를 호출하게 됩니다. (C++ 프로그래머를 위한 표현으로: 파이썬의 모든 메서드는 실질적으로 `virtual` 입니다.)

파생 클래스에서 재정의된 메서드가, 같은 이름의 베이스 클래스 메서드를 단순히 갈아치우기보다 사실은 확장하고 싶을 수 있습니다. 베이스 클래스의 메서드를 직접 호출하는 간단한 방법이 있습니다: 단지 `BaseClassName.methodname(self, arguments)` 를 호출하면 됩니다. 이것은 때로 클라이언트에게도 쓸모가 있습니다. (이것은 베이스 클래스가 전역 스코프에서 `BaseClassName` 으로 액세스 될 수 있을 때만 동작함에 주의하세요.)

파이썬에는 상속과 함께 사용할 수 있는 두 개의 내장 함수가 있습니다:

- 인스턴스의 형을 검사하려면 `isinstance()` 를 사용합니다: `isinstance(obj, int)` 는 `obj.__class__` 가 `int` 거나 `int` 에서 파생된 클래스인 경우만 `True` 가 됩니다.
- 클래스 상속을 검사하려면 `issubclass()` 를 사용합니다: `issubclass(bool, int)` 는 `True` 인데, `bool` 이 `int` 의 서브 클래스이기 때문입니다. 하지만, `issubclass(float, int)` 는 `False` 인데, `float` 는 `int` 의 서브 클래스가 아니기 때문입니다.

## 9.5.1 다중 상속

파이썬은 다중 상속의 형태도 지원합니다. 여러 개의 베이스 클래스를 갖는 클래스 정의는 이런 식입니다:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

사실, 이것보다는 약간 더 복잡합니다; 메서드 결정 순서는 `super()` 로의 협력적인 호출을 지원하기 위해 동적으로 변경됩니다. 이 접근법은 몇몇 다른 다중 상속 언어들에서 `call-next-method` 라고 알려져 있고, 단일 상속 언어들에서 발견되는 `super` 호출보다 더 강력합니다.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see `python_2.3_mro`.

## 9.6 비공개 변수

객체 내부에서만 액세스할 수 있는 “비공개” 인스턴스 변수는 파이썬에 존재하지 않습니다. 하지만, 대부분의 파이썬 코드에서 따르고 있는 규약이 있습니다: 밑줄로 시작하는 이름은 (예를 들어, `_spam`) API의

공개적이지 않은 부분으로 취급되어야 합니다 (그것이 함수, 메서드, 데이터 멤버중 무엇이건 간에). 구현 상세이고 통보 없이 변경되는 대상으로 취급되어야 합니다.

클래스-비공개 멤버들의 올바른 사례가 있으므로 (즉 서브 클래스에서 정의된 이름들과의 충돌을 피하고자), 이름 뒤섞기 (*name mangling*) 라고 불리는 메커니즘에 대한 제한된 지원이 있습니다. `__spam` 형태의 (최소 두 개의 밑줄로 시작하고, 최대 한 개의 밑줄로 끝납니다) 모든 식별자는 `__classname__spam` 로 텍스트 적으로 치환되는데, `classname` 은 현재 클래스 이름에서 앞에 오는 밑줄을 제거한 것입니다. 이 뒤섞기는 클래스 정의에 등장하는 이상, 식별자의 문법적 위치와 무관하게 수행됩니다.

### ➡ 더 보기

The private name mangling specifications for details and special cases.

이름 뒤섞기는 클래스 내부의 메서드 호출을 방해하지 않고 서브 클래스들이 메서드를 재정의할 수 있도록 하는 데 도움을 줍니다. 예를 들어:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

위의 예는 `MappingSubclass`가 `__update` 식별자를 도입하더라도 작동합니다. `Mapping` 클래스에서는 `__Mapping__update`로 `MappingSubclass` 클래스에서는 `__MappingSubclass__update`로 각각 대체 되기 때문입니다.

뒤섞기 규칙은 대체로 사고를 피하고자 설계되었다는 것에 주의하세요; 여전히 비공개로 취급되는 변수들을 액세스하거나 수정할 수 있습니다. 이것은 디버거와 같은 특별한 상황에서 쓸모 있기조차 합니다.

`exec()` 나 `eval()` 로 전달된 코드는 호출하는 클래스의 클래스 이름을 현재 클래스로 여기지 않는다는 것에 주의하세요; 이것은 `global` 문의 효과와 유사한데, 효과가 함께 바이트-컴파일된 코드로 제한됩니다. 같은 제약이 `__dict__` 를 직접 참조할 때뿐만 아니라, `getattr()`, `setattr()`, `delattr()` 에도 적용됩니다.

## 9.7 잡동사니

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. The idiomatic approach is to use `dataclasses` for this purpose:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
dept: str
salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

## 9.8 이터레이터

지금쯤 아마도 여러분은 대부분의 컨테이너 객체들을 `for` 문으로 루핑할 수 있음을 눈치챘을 것입니다:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

이런 스타일의 액세스는 명료하고, 간결하고, 편리합니다. 이터레이터를 사용하면 파이썬이 보편화하고 통합됩니다. 무대 뒤에서, `for` 문은 컨테이너 객체에 대해 `iter()` 를 호출합니다. 이 함수는 메서드 `__next__()` 를 정의하는 이터레이터 객체를 돌려주는데, 이 메서드는 컨테이너의 요소들을 한 번에 하나씩 액세스합니다. 남은 요소가 없으면, `__next__()` 는 `StopIteration` 예외를 일으켜서 `for` 루프에 종료를 알립니다. `next()` 내장 함수를 사용해서 `__next__()` 메서드를 호출할 수 있습니다; 이 예는 이 모든 것들이 어떻게 동작하는지 보여줍니다:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then

`__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

## 9.9 제너레이터

제너레이터는 이터레이터를 만드는 간단하고 강력한 도구입니다. 일반적인 함수처럼 작성되지만 값을 돌려주고 싶을 때마다 `yield` 문을 사용합니다. 제너레이터에 `next()`가 호출될 때마다, 제너레이터는 떠난 곳에서 실행을 재개합니다(모든 데이터 값들과 어떤 문장이 마지막으로 실행되었는지 기억합니다). 예는 제너레이터를 사소할 정도로 쉽게 만들 수 있음을 보여줍니다:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

또 하나의 주요 기능은 지역 변수들과 실행 상태가 호출 간에 자동으로 보관된다는 것입니다. 이것은 `self.index` 나 `self.data` 와 같은 인스턴스 변수를 사용하는 접근법에 비해 함수를 쓰기 쉽고 명료하게 만듭니다.

자동 메서드 생성과 프로그램 상태의 저장에 더해, 제너레이터가 종료할 때 자동으로 `StopIteration`을 일으킵니다. 조합하면, 이 기능들이 일반 함수를 작성하는 것만큼 이터레이터를 만들기 쉽게 만듭니다.

## 9.10 제너레이터 표현식

간단한 제너레이터는 리스트 컴프리헨션과 비슷하지만, 대괄호 대신 괄호를 사용하는 문법을 사용한 표현식으로 간결하게 코딩할 수 있습니다. 이 표현식들은 둘러싸는 함수가 제너레이터를 즉시 사용하는 상황을 위해 설계되었습니다. 제너레이터 표현식은 완전한 제너레이터 정의보다 간결하지만, 융통성은 떨어지고, 비슷한 리스트 컴프리헨션보다 메모리를 덜 쓰는 경향이 있습니다.

예:

```
>>> sum(i*i for i in range(10))           # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))   # dot product
260

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```





## 표준 라이브러리 둘러보기

### 10.1 운영 체제 인터페이스

os 모듈은 운영 체제와 상호 작용하기 위한 수십 가지 함수들을 제공합니다:

```
>>> import os
>>> os.getcwd()           # Return the current working directory
'C:\\Python313'
>>> os.chdir('/server/accesslogs') # Change current working directory
>>> os.system('mkdir today')      # Run the command mkdir in the system shell
0
```

from os import \* 대신에 import os 스타일을 사용해야 합니다. 그래야 os.open() 이 내장 open() 을 가리는 것을 피할 수 있는데, 두 함수는 아주 다르게 동작합니다.

os 와 같은 큰 모듈과 작업할 때, 내장 dir() 과 help() 함수는 대화형 도우미로 쓸모가 있습니다.

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

일상적인 파일과 디렉터리 관리 작업을 위해, shutil 모듈은 사용하기 쉬운 더 고수준의 인터페이스를 제공합니다:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

### 10.2 파일 와일드카드

glob 모듈은 디렉터리 와일드카드 검색으로 파일 목록을 만드는 함수를 제공합니다:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3 명령행 인자

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

`argparse` 모듈은 명령 줄 인자를 처리하는 더 정교한 메커니즘을 제공합니다. 다음 스크립트는 하나 이상의 파일명과 선택적으로 표시할 줄 수를 추출합니다:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

`python top.py --lines=5 alpha.txt beta.txt`를 사용하여 명령 줄에서 실행할 때, 스크립트는 `args.lines`를 5로, `args.filenames`를 `['alpha.txt', 'beta.txt']`로 설정합니다.

## 10.4 에러 출력 리디렉션과 프로그램 종료

`sys` 모듈은 `stdin`, `stdout`, `stderr` 어트리뷰트도 갖고 있습니다. 가장 마지막 것은 `stdout` 이 리디렉트 되었을 때도 볼 수 있는 경고와 에러 메시지들을 출력하는데 쓸모가 있습니다:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

스크립트를 종료하는 가장 직접적인 방법은 `sys.exit()` 를 쓰는 것입니다.

## 10.5 문자열 패턴 매칭

`re` 모듈은 고급 문자열 처리를 위한 정규식 도구들을 제공합니다. 복잡한 매칭과 조작을 위해, 정규식은 간결하고 최적화된 솔루션을 제공합니다:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

단지 간단한 기능만 필요한 경우에는, 문자열 메서드들이 선호되는데 읽기 쉽고 디버깅이 쉽기 때문입니다:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6 수학

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

`random` 모듈은 무작위 선택을 할 수 있는 도구들을 제공합니다:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()                  # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)              # random integer chosen from range(6)
4
```

`statistics` 모듈은 수치 데이터의 기본적인 통계적 특성들을 (평균, 중간값, 분산, 등등) 계산합니다.

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

SciPy 프로젝트 <<https://scipy.org>> 는 다른 수치 계산용 모듈들을 많이 갖고 있습니다.

## 10.7 인터넷 액세스

인터넷을 액세스하고 인터넷 프로토콜들을 처리하는 많은 모듈이 있습니다. 가장 간단한 두 개는 URL 에서 데이터를 읽어오는 `urllib.request` 와 메일을 보내는 `smtplib` 입니다:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()                # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())            # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """
>>> server.quit()
```

(두 번째 예는 localhost 에서 메일 서버가 실행되고 있어야 한다는 것에 주의하세요.)

## 10.8 날짜와 시간

datetime 모듈은 날짜와 시간을 조작하는 클래스들을 제공하는데, 간단한 방법과 복잡한 방법 모두 제공합니다. 날짜와 시간 산술이 지원되지만, 구현의 초점은 출력 포매팅과 조작을 위해 효율적으로 멤버를 추출하는 데에 맞춰져 있습니다. 모듈은 시간대를 고려하는 객체들도 지원합니다.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9 데이터 압축

일반적인 데이터 보관 및 압축 형식들을 다음과 같은 모듈들이 직접 지원합니다: zlib, gzip, bz2, lzma, zipfile, tarfile.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10 성능 측정

일부 파이썬 사용자들은 같은 문제에 대한 다른 접근법들의 상대적인 성능을 파악하는데 깊은 관심을 두고 있습니다. 파이썬은 이런 질문들에 즉시 답을 주는 측정 도구를 제공합니다.

예를 들어, 인자들을 맞교환하는 전통적인 방식 대신에, 튜플 패킹과 언 패킹을 사용하고자 하는 유혹을 느낄 수 있습니다. timeit 모듈은 적당한 성능 이점을 신속하게 보여줍니다:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

timeit의 정밀도와는 대조적으로, profile과 pstats 모듈은 큰 블록의 코드에서 시간 임계 섹션을 식별하기 위한 도구들을 제공합니다.

## 10.11 품질 관리

고품질의 소프트웨어를 개발하는 한 가지 접근법은 개발되는 각 함수에 대한 테스트를 작성하고, 그것들을 개발 프로세스 중에 자주 실행하는 것입니다.

doctest 모듈은 모듈을 훑어보고 프로그램의 독스트링들에 내장된 테스트들을 검사하는 도구를 제공합니다. 테스트 만들기는 평범한 호출을 그 결과와 함께 독스트링으로 복사해서 붙여넣기를 하는 수준으로 간단해집니다. 사용자에게 예제를 함께 제공해서 설명서를 개선하고, doctest 모듈이 설명서에서 코드가 여전히 사실인지 확인하도록 합니다.

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

unittest 모듈은 doctest 모듈만큼 쉬운 것은 아니지만, 더욱 포괄적인 테스트 집합을 별도의 파일로 관리할 수 있게 합니다:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12 배터리가 포함됩니다

파이썬은 “배터리가 포함됩니다” 철학을 갖고 있습니다. 이는 더 큰 패키지의 정교하고 강력한 기능을 통해 가장 잘 나타납니다. 예를 들어:

- The `xmlrpc.client` and `xmlrpc.server` modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- email 패키지는 MIME 및 기타 [RFC 2822](#) 기반 메시지 문서를 포함하는 전자 메일 메시지를 관리하기 위한 라이브러리입니다. 실제로 메시지를 보내고 받는 `smtpplib`와 `poplib`와는 달리, email 패키지

는 복잡한 메시지 구조 (첨부 파일 포함) 를 작성하거나 해독하고 인터넷 인코딩과 헤더 프로토콜을 구현하기 위한 완벽한 도구 상자를 가지고 있습니다.

- `json` 패키지는 널리 사용되는 데이터 교환 형식을 파싱하기 위한 강력한 지원을 제공합니다. `csv` 모듈은 데이터베이스와 스프레드시트에서 일반적으로 지원되는 쉼표로 구분된 값 형식으로 파일을 직접 읽고 쓸 수 있도록 지원합니다. `XML` 처리는 `xml.etree.ElementTree`, `xml.dom` 및 `xml.sax` 패키지에 의해 지원됩니다. 이러한 모듈과 패키지를 함께 사용하면 파이썬 응용 프로그램과 다른 도구 간의 데이터 교환이 크게 단순해집니다.
- `sqlite3` 모듈은 `SQLite` 데이터베이스 라이브러리의 래퍼인데, 약간 비표준 `SQL` 구문을 사용하여 업데이트되고 액세스될 수 있는 퍼시스턴트 데이터베이스를 제공합니다.
- 국제화는 `gettext`, `locale`, 그리고 `codecs` 패키지를 포함한 많은 모듈에 의해 지원됩니다.

## 표준 라이브러리 둘러보기 — 2부

이 두 번째 둘러보기는 전문 프로그래밍 요구 사항을 지원하는 고급 모듈을 다루고 있습니다. 이러한 모듈은 작은 스크립트에서는 거의 사용되지 않습니다.

## 11.1 출력 포매팅

`reprlib` 모듈은 크거나 깊게 중첩된 컨테이너의 축약된 디스플레이를 위해 커스터마이징된 `repr()`의 버전을 제공합니다:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

`pprint` 모듈은 인터프리터가 읽을 수 있는 방식으로 내장 객체나 사용자 정의 객체를 인쇄하는 것을 보다 정교하게 제어할 수 있게 합니다. 결과가 한 줄보다 길면 “예쁜 프린터”가 줄 바꿈과 들여쓰기를 추가하여 데이터 구조를 보다 명확하게 나타냅니다:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...           'yellow'], 'blue']]
...
>>> pprint.pprint(t, width=30)
[[['black', 'cyan',
    'white',
    ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

`textwrap` 모듈은 텍스트의 문단을 주어진 화면 너비에 맞게 포맷합니다:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale 모듈은 문화권 특정 데이터 포맷의 데이터베이스에 액세스합니다. locale의 format 함수의 grouping 어트리뷰트는 그룹 구분 기호로 숫자를 포맷팅하는 직접적인 방법을 제공합니다:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()           # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                               conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

## 11.2 템플릿

string 모듈은 다재다능한 Template 클래스를 포함하고 있는데, 최종 사용자가 편집하기에 적절한 단순한 문법을 갖고 있습니다. 따라서 사용자는 응용 프로그램을 변경하지 않고도 응용 프로그램을 커스터마이징할 수 있습니다.

형식은 \$와 유효한 파이썬 식별자(영숫자와 밑줄)로 만들어진 자리표시자 이름을 사용합니다. 중괄호를 사용하여 자리표시자를 둘러싸면 공백없이 영숫자가 뒤따르도록 할 수 있습니다. \$\$ 을 쓰면 하나의 이스케이프 된 \$ 를 만듭니다:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

substitute() 메서드는 자리표시자가 딕셔너리나 키워드 인자로 제공되지 않을 때 KeyError 를 일으킵니다. 메일 병합 스타일 응용 프로그램의 경우 사용자가 제공한 데이터가 불완전할 수 있으며 safe\_substitute() 메서드가 더 적절할 수 있습니다. 데이터가 누락된 경우 자리표시자를 변경하지 않습니다:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template 서브 클래스는 사용자 정의 구분자를 지정할 수 있습니다. 예를 들어 사진 브라우저를 위한 일괄 이름 바꾸기 유틸리티는 현재 날짜, 이미지 시퀀스 번호 또는 파일 형식과 같은 자리표시자에 백분율 기호를 사용하도록 선택할 수 있습니다:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```

...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg

```

템플릿의 또 다른 응용은 다중 출력 형식의 세부 사항에서 프로그램 논리를 분리하는 것입니다. 이렇게 하면 XML 파일, 일반 텍스트 보고서 및 HTML 웹 보고서에 대한 커스텀 템플릿을 치환할 수 있습니다.

## 11.3 바이너리 데이터 레코드 배치 작업

struct 모듈은 가변 길이 바이너리 레코드 형식으로 작업하기 위한 pack() 과 unpack() 함수를 제공합니다. 다음 예제는 zipfile 모듈을 사용하지 않고 ZIP 파일의 헤더 정보를 루핑하는 법을 보여줍니다. 팩 코드 "H" 와 "I" 는 각각 2바이트와 4바이트의 부호 없는 숫자를 나타냅니다. "<" 는 표준 크기이면서 리틀 엔디안 바이트 순서를 가짐을 나타냅니다:

```

import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):
    # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size    # skip to the next header

```

## 11.4 다중 스레딩

스레딩은 차례로 종속되지 않는 작업을 분리하는 기술입니다. 스레드는 다른 작업이 백그라운드에서 실행되는 동안 사용자 입력을 받는 응용 프로그램의 응답을 향상하는 데 사용할 수 있습니다. 관련된 사용 사례는 다른 스레드의 계산과 병렬로 I/O를 실행하는 경우입니다.

다음 코드는 메인 프로그램이 계속 실행되는 동안 고수준 threading 모듈이 백그라운드에서 작업을 어떻게 수행할 수 있는지 보여줍니다:

```
import threading, zipfile
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')

```

다중 스레드 응용 프로그램의 가장 큰 문제점은 데이터 또는 다른 자원을 공유하는 스레드를 조정하는 것입니다. 이를 위해 `threading` 모듈은 락, 이벤트, 조건 변수 및 세마포어를 비롯한 많은 수의 동기화 기본 요소를 제공합니다.

이러한 도구는 강력하지만, 사소한 설계 오류로 인해 재현하기 어려운 문제가 발생할 수 있습니다. 따라서, 작업 조정에 대한 선호되는 접근 방식은 자원에 대한 모든 액세스를 단일 스레드에 집중시킨 다음 `queue` 모듈을 사용하여 해당 스레드에 다른 스레드의 요청을 제공하는 것입니다. 스레드 간 통신 및 조정을 위한 `Queue` 객체를 사용하는 응용 프로그램은 설계하기 쉽고, 읽기 쉽고, 신뢰성이 높습니다.

## 11.5 로깅

`logging` 모듈은 완전한 기능을 갖춘 유연한 로깅 시스템을 제공합니다. 가장 단순한 경우, 로그 메시지는 파일이나 `sys.stderr`로 보내집니다:

```

import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')

```

그러면 다음과 같은 결과가 출력됩니다:

```

WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down

```

기본적으로 정보 및 디버깅 메시지는 표시되지 않고 출력은 표준 에러로 보내집니다. 다른 출력 옵션에는 전자 메일, 데이터 그래프, 소켓 또는 HTTP 서버를 통한 메시지 라우팅이 포함됩니다. 새로운 필터는 메시지 우선순위에 따라 다른 라우팅을 선택할 수 있습니다: `DEBUG`, `INFO`, `WARNING`, `ERROR`, 그리고 `CRITICAL`.

로깅 시스템은 파이썬에서 직접 구성하거나, 응용 프로그램을 변경하지 않고 사용자 정의 로깅을 위해 사용자가 편집할 수 있는 설정 파일에서 로드할 수 있습니다.

## 11.6 약한 참조

파이썬은 자동 메모리 관리 (대부분 객체에 대한 참조 횟수 추적 및 순환을 제거하기 위한 가비지 수거)를 수행합니다. 메모리는 마지막 참조가 제거된 직후에 해제됩니다.

이 접근법은 대부분의 응용 프로그램에서 잘 작동하지만, 때로는 다른 것들에 의해 사용되는 동안에만 객체를 추적해야 할 필요가 있습니다. 불행하게도, 단지 그것들을 추적하는 것만으로도 그들을 영구적으로 만드는 참조를 만듭니다. `weakref` 모듈은 참조를 만들지 않고 객체를 추적할 수 있는 도구를 제공합니다. 객체가 더 필요하지 않으면 `weakref` 테이블에서 객체가 자동으로 제거되고 `weakref` 객체에 대한 콜백이 트리거됩니다. 일반적인 응용에는 만드는 데 비용이 많이 드는 개체 캐싱이 포함됩니다:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                                # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a                          # does not create a reference
>>> d['primary']                              # fetch the object if it is still alive
10
>>> del a                                    # remove the one reference
>>> gc.collect()                            # run garbage collection right away
0
>>> d['primary']                             # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                             # entry was automatically removed
  File "C:/python313/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

## 11.7 리스트 작업 도구

내장 리스트 형으로 많은 데이터 구조 요구를 충족시킬 수 있습니다. 그러나 때로는 다른 성능 상충 관계가 있는 대안적 구현이 필요할 수도 있습니다.

The `array` module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
    unsearched.append(m)
```

대안적 리스트 구현 외에도 라이브러리는 정렬된 리스트를 조작하는 함수들이 있는 `bisect` 모듈과 같은 다른 도구를 제공합니다:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

`heapq` 모듈은 일반 리스트를 기반으로 힙을 구현하는 함수를 제공합니다. 가장 값이 작은 항목은 항상 위치 0에 유지됩니다. 이것은 가장 작은 요소에 반복적으로 액세스하지만, 전체 목록 정렬을 실행하지 않으려는 응용에 유용합니다:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

## 11.8 Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- 정확한 10진수 표현이 필요한 금융 응용 및 기타 용도,
- 정밀도 제어,
- 법적 또는 규제 요구 사항을 충족하는 반올림 제어,
- 유효숫자 추적, 또는
- 사용자가 결과가 손으로 계산한 것과 일치 할 것으로 기대하는 응용.

예를 들어, 70센트 전화 요금에 대해 5% 세금을 계산하면, 십진 부동 소수점 및 이진 부동 소수점에 다른 결과가 나타납니다. 결과를 가장 가까운 센트로 반올림하면 차이가 드러납니다:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

`Decimal` 결과는 끝에 붙는 0을 유지하며, 두 개의 유효숫자를 가진 피승수로부터 네 자리의 유효숫자를 자동으로 추론합니다. `Decimal`은 손으로 한 수학을 재현하고 이진 부동 소수점이 십진수를 정확하게 표현할 수 없을 때 발생할 수 있는 문제를 피합니다.

정확한 표현은 `Decimal` 클래스가 이진 부동 소수점에 적합하지 않은 모듈로 계산과 동등성 검사를 수행할 수 있도록 합니다:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

`decimal` 모듈은 필요한 만큼의 정밀도로 산술을 제공합니다:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857')
```



## 가상 환경 및 패키지

## 12.1 소개

파이썬 응용 프로그램은 종종 표준 라이브러리의 일부로 제공되지 않는 패키지와 모듈을 사용합니다. 응용 프로그램에 특정 버전의 라이브러리가 필요할 수 있는데, 응용 프로그램에 특정 버그가 수정된 버전이 필요하거나, 라이브러리 인터페이스의 구식 버전을 사용하여 응용 프로그램을 작성할 수도 있기 때문입니다.

즉, 하나의 파이썬 설치에 모든 응용 프로그램의 요구 사항을 충족시키는 것이 불가능할 수도 있습니다. 응용 프로그램 A에 특정 모듈의 버전 1.0이 필요하지만, 응용 프로그램 B에 버전 2.0이 필요한 경우, 요구 사항이 충돌하고, 버전 1.0 또는 2.0을 설치하면 어느 한 응용 프로그램은 실행할 수 없게 됩니다.

이 문제에 대한 해결책은 **가상 환경**을 만드는 것입니다. 이 가상 환경은 특정 버전 파이썬 설치와 여러 추가 패키지를 포함하는 완비된 디렉터리 트리입니다.

서로 다른 응용 프로그램은 서로 다른 가상 환경을 사용할 수 있습니다. 앞서 본 상충하는 요구 사항의 예를 해결하기 위해, 응용 프로그램 A에는 버전 1.0이 설치된 자체 가상 환경이 있고, 응용 프로그램 B에는 버전 2.0이 있는 다른 가상 환경이 있을 수 있습니다. 응용 프로그램 B에서 라이브러리를 버전 3.0으로 업그레이드해야 하는 경우, 응용 프로그램 A의 환경에 영향을 미치지 않습니다.

## 12.2 가상 환경 만들기

The module used to create and manage virtual environments is called `venv`. `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

가상 환경을 만들려면, 원하는 디렉터를 결정하고, `venv` 모듈을 스크립트로 실행하는데 디렉터리 경로를 명령행 인자로 전달합니다:

```
python -m venv tutorial-env
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter and various supporting files.

가상 환경의 일반적인 디렉터리 위치는 `.venv`입니다. 이 이름은 디렉터리가 보통 셸에서 숨겨져 있도록 하므로, 디렉터리가 존재하는 이유를 설명하는 이름을 제공하면서도 방해받지 않습니다. 또한 일부 툴링(tooling)이 지원하는 `.env` 환경 변수 정의 파일과의 충돌을 방지합니다.

가상 환경을 만들었으면, 가상 환경을 활성화할 수 있습니다.

윈도우에서 이렇게 실행합니다:

```
tutorial-env\Scripts\activate
```

Unix 또는 MacOS에서 이렇게 실행합니다:

```
source tutorial-env/bin/activate
```

(이 스크립트는 `bash` 셸을 위해 작성된 것으로, `csh` 또는 `fish` 셸을 사용하는 경우에는, 대신 `activate.csh` 와 `activate.fish` 스크립트를 사용해야 합니다.)

가상 환경을 활성화하면, 셸의 프롬프트가 변경되어 사용 중인 가상 환경을 보여주고, 환경을 수정하여 `python` 을 실행하면 특정 버전의 파이썬이 실행되도록 합니다. 예를 들어:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36)
...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
 '~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

To deactivate a virtual environment, type:

```
deactivate
```

into the terminal.

## 12.3 pip로 패키지 관리하기

You can install, upgrade, and remove packages using a program called **pip**. By default `pip` will install packages from the [Python Package Index](#). You can browse the Python Package Index by going to it in your web browser.

`pip` has a number of subcommands: “install”, “uninstall”, “freeze”, etc. (Consult the installing-index guide for complete documentation for `pip`.)

패키지 이름을 지정하여 최신 버전의 패키지를 설치할 수 있습니다:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

패키지 이름 뒤에 `==` 과 버전 번호를 붙여 특정 버전의 패키지를 설치할 수도 있습니다:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you re-run this command, `pip` will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run `python -m pip install --upgrade requests` to upgrade the package to the latest version:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
```

(다음 페이지에 계속)



(이전 페이지에서 계속)

```
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`python -m pip uninstall` followed by one or more package names will remove the packages from the virtual environment.

`python -m pip show` will display information about a particular package:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` will display all of the packages installed in the virtual environment:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` will produce a similar list of the installed packages, but the output uses the format that `python -m pip install` expects. A common convention is to put this list in a `requirements.txt` file:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

`requirements.txt` 는 버전 제어에 커밋되어 응용 프로그램 일부로 제공될 수 있습니다. 사용자는 `install -r` 로 모든 필요한 패키지를 설치할 수 있습니다:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` has many more options. Consult the installing-index guide for complete documentation for `pip`. When you've written a package and want to make it available on the Python Package Index, consult the [Python packaging user](#)

guide.

## CHAPTER 13

---

### 이제 뭘 하지?

---

이 자습서를 읽어서 아마도 파이썬 사용에 관한 관심이 높아졌을 것입니다 — 실제 문제를 해결하기 위해 파이썬을 적용하려고 열망해야 합니다. 더 배우려면 어디로 가야 할까?

이 자습서는 파이썬의 문서 세트의 일부입니다. 세트의 다른 문서는 다음과 같습니다:

- **library-index:**

표준 라이브러리의 형, 함수 및 모듈에 대한 완전한 (비록 딱딱하지만) 레퍼런스 자료를 제공하는 이 설명서를 탐색해야 합니다. 표준 파이썬 배포판에는 추가 코드가 많이 포함되어 있습니다. 유닉스 우편함을 읽고, HTTP를 통해 문서를 검색하고, 난수를 만들고, 명령행 옵션을 파싱하고, 데이터를 압축하고, 기타 많은 작업을 수행하는 모듈이 있습니다. 라이브러리 레퍼런스를 훑어보면 어떤 것이 있는지 알 수 있습니다.

- **installing-index** 는 다른 파이썬 사용자가 작성한 추가 모듈을 설치하는 방법을 설명합니다.

- **reference-index:** 파이썬의 문법과 의미에 대한 자세한 설명. 읽기에 부담스럽지만, 언어 자체에 대한 완전한 안내서로서 유용합니다.

기타 파이썬 자료:

- <https://www.python.org>: 주요 파이썬 웹 사이트. 여기에는 코드, 문서 및 웹에 있는 파이썬 관련 페이지들에 대한 포인터가 들어 있습니다.
- <https://docs.python.org>: 파이썬의 설명서에 빠르게 액세스할 수 있습니다.
- <https://pypi.org>: 이전에 치즈 가게 (Cheese Shop)로도 불렸던<sup>1</sup> 파이썬 패키지 인덱스는 내려받을 수 있는 사용자 제작 파이썬 모듈의 색인입니다. 코드를 배포하기 시작하면 다른 사람들이 찾을 수 있도록 여기에 코드를 등록할 수 있습니다.
- <https://code.activestate.com/recipes/langs/python/>: 파이썬 요리책 (Python Cookbook)은 많은 코드 예제, 더 큰 모듈 및 유용한 스크립트 모음입니다. 특히 주목할만한 공헌들을 Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3)이라는 제목의 책에 모았습니다.
- <https://pyvideo.org> 는 컨퍼런스 및 사용자 그룹 회의에서 파이썬 관련 비디오에 대한 링크들을 수집합니다.
- <https://scipy.org>: Scientific Python 프로젝트에는 빠른 배열 계산 및 조작을 위한 모듈들과 선형 대수, 푸리에 변환, 비선형 솔버, 난수 분포, 통계 분석 등과 같은 여러 가지 패키지들이 포함되어 있습니다.

---

<sup>1</sup> “Cheese Shop”은 Monty Python의 스케치입니다: 고객이 치즈 가게에 들어가지만, 찾는 치즈가 무엇이건, 점원은 없다고 말합니다.

파이썬 관련 질문 및 문제 보고의 경우, 뉴스 그룹 *comp.lang.python*에 게시하거나 [python-list@python.org](mailto:python-list@python.org)의 메일링 리스트로 보낼 수 있습니다. 뉴스 그룹과 메일링 리스트는 게이트웨이로 연결되어 있으므로 하나에 게시된 메시지는 자동으로 다른 그룹으로 전달됩니다. 하루에 수백 건의 게시물이 올라옵니다. 질문하고, 질문에 답변하고, 새로운 기능을 제안하고, 새로운 모듈을 발표합니다. 메일링 리스트 저장소는 <https://mail.python.org/pipermail/>에 있습니다.

게시하기 전에 자주 나오는 질문들 (FAQ라고도 한다) 목록을 확인해야 합니다. FAQ는 반복적으로 나타나는 많은 질문에 대한 답을 제공하며, 이미 여러분의 문제에 대한 해결 방법을 담고 있을 수 있습니다.

---

## 대화형 입력 편집 및 히스토리 치환

---

일부 파이썬 인터프리터 버전은 Korn 셸 및 GNU Bash 셸에 있는 기능과 유사하게 현재 입력 줄 편집 및 히스토리 치환을 지원합니다. 이는 다양한 스타일의 편집을 지원하는 [GNU Readline](#) 라이브러리를 사용하여 구현됩니다. 이 라이브러리에는 자체 설명서가 있고, 여기에서 반복하지는 않습니다.

### 14.1 탭 완성 및 히스토리 편집

변수와 모듈 이름의 완성은 인터프리터 시작 시 자동으로 활성화 되어서 Tab 키가 완료 기능을 호출합니다; 파이썬 명령문 이름, 현재 지역 변수 및 사용 가능한 모듈 이름을 찾습니다. `string.a`와 같은 점으로 구분된 표현식의 경우, 표현식을 마지막 '.' 까지 값을 구한 다음, 결과 객체의 어트리뷰트로 완성을 제안합니다. `__getattr__()` 메서드를 가진 객체가 표현식의 일부면 응용 프로그램이 정의한 코드를 실행할 수 있음에 주의해야 합니다. 기본 설정은 사용자 디렉터리에 `.python_history` 라는 파일로 히스토리를 저장합니다. 다음 대화형 인터프리터 세션에서 히스토리를 다시 사용할 수 있습니다.

### 14.2 대화형 인터프리터 대안

이 기능은 이전 버전의 인터프리터에 비교했을 때 엄청난 발전입니다; 그러나, 몇 가지 아쉬움이 남습니다: 이어지는 줄에 적절한 들여쓰기가 제안된다면 좋을 것입니다 (구문 분석기는 다음에 들여쓰기 (INDENT) 토큰이 필요한지 알 수 있습니다). 완성 메커니즘은 인터프리터의 심볼 테이블을 사용할 수 있습니다. 일치하는 괄호, 따옴표 등을 검사하는 (또는 제안하는) 명령도 유용할 것입니다.

꽤 오랫동안 사용됐던 개선된 대화형 인터프리터는 [IPython](#) 인데, 탭 완성, 객체 탐색 및 고급 히스토리 관리 기능을 갖추고 있습니다. 또한, 철저하게 커스터마이징해서 다른 응용 프로그램에 내장할 수 있습니다. 비슷한 또 다른 개선된 대화형 환경은 [bpython](#) 입니다.



---

## Floating-Point Arithmetic: Issues and Limitations

---

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the **decimal** fraction 0.625 has value  $6/10 + 2/100 + 5/1000$ , and in the same way the **binary** fraction 0.101 has value  $1/2 + 0/4 + 1/8$ . These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

불행히도, 대부분의 십진 소수는 정확하게 이진 소수로 표현될 수 없습니다. 결과적으로, 일반적으로 입력하는 십진 부동 소수점 숫자가 실제로 기계에 저장될 때는 이진 부동 소수점 수로 근사될 뿐입니다.

이 문제는 먼저 밑 10에서 따져보는 것이 이해하기 쉽습니다. 분수  $1/3$ 을 생각해봅시다. 이 값을 십진 소수로 근사할 수 있습니다:

```
0.3
```

또는, 더 정확하게,

```
0.33
```

또는, 더 정확하게,

```
0.333
```

등등. 아무리 많은 자릿수를 적어도 결과가 정확하게  $1/3$ 이 될 수 없지만, 점점 더  $1/3$ 에 가까운 근사치가 됩니다.

같은 방식으로, 아무리 많은 자릿수의 숫자를 사용해도, 십진수 0.1은 이진 소수로 정확하게 표현될 수 없습니다. 이진법에서,  $1/10$ 은 무한히 반복되는 소수입니다

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

유한 한 비트 수에서 멈추면, 근삿값을 얻게 됩니다. 오늘날 대부분 기계에서, float는 이진 분수로 근사되는데, 최상위 비트로부터 시작하는 53비트를 분자로 사용하고, 2의 거듭제곱 수를 분모로 사용합니다.  $1/10$ 의 경우, 이진 분수는  $3602879701896397 / 2^{55}$ 인데, 실제 값  $1/10$ 과 거의 같지만 정확히 같지는 않습니다.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead:

```
>>> 1 / 10
0.1
```

인쇄된 결과가 정확히 1/10인 것처럼 보여도, 실제 저장된 값은 가장 가까운 표현 가능한 이진 소수임을 기억하세요.

흥미롭게도, 가장 가까운 근사 이진 소수를 공유하는 여러 다른 십진수가 있습니다. 예를 들어, 0.1 과 0.10000000000000001 및 0.1000000000000000055511151231257827021181583404541015625 는 모두  $3602879701896397 / 2^{55}$  로 근사 됩니다. 이 십진 값들이 모두 같은 근사값을 공유하기 때문에 `eval(repr(x)) == x` 불변을 그대로 유지하면서 그중 하나를 표시할 수 있습니다.

역사적으로, 파이썬 프롬프트와 내장 `repr()` 함수는 유효 숫자 17개의 숫자인 0.10000000000000001 을 선택합니다. 파이썬 3.1 부터, 이제 파이썬(대부분 시스템에서)이 가장 짧은 것을 선택할 수 있으며, 단순히 0.1 만 표시합니다.

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```
>>> format(math.pi, '.12g') # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

이것이, 진정한 의미에서, 환영임을 깨닫는 것이 중요합니다: 여러분은 단순히 진짜 기껏값의 표시를 받을림하고 있습니다.

One illusion may beget another. For example, since 0.1 is not exactly 1/10, summing three values of 0.1 may not yield exactly 0.3, either:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Also, since the 0.1 cannot get any closer to the exact value of 1/10 and 0.3 cannot get any closer to the exact value of 3/10, then pre-rounding with `round()` function cannot help:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Though the numbers cannot be made closer to their intended exact values, the `math.isclose()` function can be useful for comparing inexact values:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatively, the `round()` function can be used to compare rough approximations:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```



Binary floating-point arithmetic holds many surprises like this. The problem with “0.1” is explained in precise detail below, in the “Representation Error” section. See [Examples of Floating Point Problems](#) for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see [The Perils of Floating Point](#) for a more complete account of other common surprises.

As that says near the end, “there are no easy answers.” Still, don’t be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in  $2^{53}$  per operation. That’s more than adequate for most tasks, but you do need to keep in mind that it’s not decimal arithmetic and that every float operation can suffer a new rounding error.

병리학적 경우가 존재하지만, 무심히 부동 소수점 산술을 사용하는 대부분은, 단순히 최종 결과를 기대하는 자릿수로 반올림해서 표시하면 기대하는 결과를 보게 될 것입니다. 보통 `str()` 만으로도 충분하며, 더 세밀하게 제어하려면 `formatstrings` 에서 `str.format()` 메서드의 포맷 지정자를 보세요.

정확한 십진 표현이 필요한 사용 사례의 경우, 회계 응용 프로그램 및 고정밀 응용 프로그램에 적합한 십진 산술을 구현하는 `decimal` 모듈을 사용해 보세요.

정확한 산술의 또 다른 형태는 유리수를 기반으로 산술을 구현하는 `fractions` 모듈에 의해 지원됩니다 (따라서  $1/3$ 과 같은 숫자는 정확하게 나타낼 수 있습니다).

If you are a heavy user of floating-point operations you should take a look at the NumPy package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a float as a fraction:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Since the ratio is exact, it can be used to losslessly recreate the original value:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

The `float.hex()` method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the float value exactly:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

표현이 정확하므로, 파이썬의 다른 버전 에 걸쳐 값을 신뢰성 있게 이식하고 (플랫폼 독립성), 같은 형식을 지원하는 다른 언어(자바나 C99 같은)와 데이터를 교환하는 데 유용합니다.

Another helpful tool is the `sum()` function which helps mitigate loss-of-precision during summation. It uses extended precision for intermediate rounding steps as values are added onto a running total. That can make a difference in overall accuracy so that the errors do not accumulate to the point where they affect the final total:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

The `math.fsum()` goes further and tracks all of the “lost digits” as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr))) # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr) # Single rounding
8.042173697819788e-13
>>> sum(arr) # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x # Multiple roundings in standard precision
...
>>> total # Straight addition has no correct digits!
-0.0051575902860057365
```

## 15.1 표현 오류

이 섹션에서는 “0.1” 예제를 자세히 설명하고, 이러한 사례에 대한 정확한 분석을 여러분이 직접 수행하는 방법을 보여줍니다. 이진 부동 소수점 표현에 대한 기본 지식이 있다고 가정합니다.

표현 오류 (*Representation error*) 는 일부 (실제로는, 대부분의) 십진 소수가 이진 (밑 2) 소수로 정확하게 표현될 수 없다는 사실을 나타냅니다. 이것이 파이썬 (또는 펄, C, C++, 자바, 포트란 및 기타 여러 언어) 이 종종 여러분이 기대하는 정확한 십진수를 표시하지 않는 주된 이유입니다.

Why is that?  $1/10$  is not exactly representable as a binary fraction. Since at least 2000, almost all machines use IEEE 754 binary floating-point arithmetic, and almost all platforms map Python floats to IEEE 754 binary64 “double precision” values. IEEE 754 binary64 values contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form  $J/2^N$  where  $J$  is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~ J / (2**N)
```

를

```
J ~ 2**N / 10
```

and recalling that  $J$  has exactly 53 bits (is  $\geq 2^{52}$  but  $< 2^{53}$ ), the best value for  $N$  is 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
True
```

That is, 56 is the only value for  $N$  that leaves  $J$  with exactly 53 bits. The best possible value for  $J$  is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to  $1/10$  in IEEE 754 double precision is:

```
7205759403792794 / 2 ** 56
```

분자와 분모를 둘로 나누면 다음과 같이 약분됩니다:

```
3602879701896397 / 2 ** 55
```

올림을 했기 때문에, 이것은 실제로 1/10 보다 약간 크다는 것에 유의하세요; 내림을 했다면, 몫이 1/10 보다 약간 작아졌을 것입니다. 그러나 어떤 경우에도 정확하게 1/10 일 수는 없습니다!

So the computer never “sees” 1/10: what it sees is the exact fraction given above, the best IEEE 754 double approximation it can get:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

If we multiply that fraction by 10\*\*55, we can see the value out to 55 decimal digits:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
10000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the decimal value 0.10000000000000000055511151231257827021181583404541015625. Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

The fractions and decimal modules make these calculations easy:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.10000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17f')
'0.10000000000000001'
```



## 16.1 대화형 모드

There are two variants of the interactive *REPL*. The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with `curses` support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see `using-on-controlling-color` for details. Function keys provide some additional functionality. `F1` enters the interactive help browser `pydoc`. `F2` allows for browsing command-line history with neither output nor the `>>` and `...` prompts. `F3` enters “paste mode”, which makes pasting larger blocks of code easier. Press `F3` to return to the regular prompt.

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the `PYTHON_BASIC_REPL` environment variable.

### 16.1.1 에러 처리

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit status; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

기본 또는 보조 프롬프트에 인터럽트 문자 (일반적으로 `Control-C` 또는 `Delete`)를 입력하면 입력을 취소하고 기본 프롬프트로 돌아갑니다.<sup>1</sup> 명령어가 실행되는 동안 인터럽트를 입력하면 `try` 문에 의해 처리될 수 있는 `KeyboardInterrupt` 예외가 발생합니다.

### 16.1.2 실행 가능한 파이썬 스크립트

BSD 스타일의 유닉스 시스템에서 파이썬 스크립트는 셸 스크립트처럼 직접 실행할 수 있게 만들 수 있습니다. 다음과 같은 줄

```
#!/usr/bin/env python3
```

<sup>1</sup> GNU Readline 패키지에 있는 문제가 이것을 방해할 수 있습니다.

(인터프리터가 사용자의 `PATH`에 있다고 가정할 때)을 스크립트의 시작 부분에 넣고 파일에 실행 가능 모드를 줍니다. `#!`는 반드시 파일의 처음 두 문자여야 합니다. 일부 플랫폼에서는 이 첫 번째 줄이 유닉스 스타일의 줄 종료(`'\n'`)로 끝나야 하며, 윈도우 줄 종료(`'\r\n'`)는 허락되지 않습니다. 파이썬에서 해시, 또는 파운드, 문자 `'#'`는 주석을 시작하는 데 사용됩니다.

스크립트는 `chmod` 명령을 사용하여 실행 가능한 모드, 또는 권한, 을 부여받을 수 있습니다.

```
$ chmod +x myscript.py
```

윈도우 시스템에서는 “실행 가능 모드”라는 개념이 없습니다. 파이썬 설치 프로그램은 `.py` 파일을 `python.exe`와 자동으로 연결하여, 파이썬 파일을 이중 클릭하면 스크립트로 실행합니다. 확장자는 `.pyw` 일 수도 있습니다. 이 경우, 일반적으로 나타나는 콘솔 창은 표시되지 않습니다.

### 16.1.3 대화형 시작 파일

파이썬을 대화형으로 사용할 때, 종종 인터프리터가 시작될 때마다 실행되는 표준 명령들이 있으면 편리합니다. `PYTHONSTARTUP` 환경 변수를 시작 명령이 들어있는 파일 이름으로 설정하면 됩니다. 이것은 유닉스 셸의 `.profile` 기능과 유사합니다.

이 파일은 대화형 세션에서만 읽히며, 파이썬이 스크립트에서 명령을 읽을 때나, `/dev/tty`가 명령의 명시적 소스인 경우(대화형 세션처럼 동작한다)에는 읽지 않습니다. 대화형 명령이 실행되는 같은 이름 공간에서 실행되므로, 이 파일에서 정의하거나 임포트하는 객체들을 대화형 세션에서 정규화하지 않은 이름으로 사용할 수 있습니다. 이 파일에서 `sys.ps1` 및 `sys.ps2` 프롬프트를 변경할 수도 있습니다.

현재 디렉터리에서 추가 시작 파일을 읽으려면, 전역 시작 파일에서 `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`와 같은 코드를 사용해서 프로그램할 수 있습니다. 스크립트에서 시작 파일을 사용하려면 스크립트에서 명시적으로 수행해야 합니다:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

### 16.1.4 커스터마이제이션 모듈

Python provides two hooks to let you customize it: `sitecustomize` and `usercustomize`. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

이제 그 디렉터리에 `usercustomize.py`라는 이름의 파일을 만들고 원하는 것들을 넣을 수 있습니다. 자동 임포트를 비활성화하는 `-s` 옵션으로 시작하지 않는 한, 이 파일은 모든 파이썬 실행에 영향을 줍니다.

`sitecustomize` works in the same way, but is typically created by an administrator of the computer in the global site-packages directory, and is imported before `usercustomize`. See the documentation of the `site` module for more details.

&gt;&gt;&gt;

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

...

다음과 같은 것들을 가리킬 수 있습니다:

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

#### abstract base class (추상 베이스 클래스)

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

#### annotation (어노테이션)

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

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

이 기능을 설명하는 **변수 어노테이션**, **함수 어노테이션**, **PEP 484**, **PEP 526**을 참조하세요. 어노테이션 작업에 대한 모범 사례는 `annotations-howto`를 참조하세요.

#### argument (인자)

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

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

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

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

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

인자는 함수 바디의 이름 붙은 지역 변수에 대입됩니다. 이 대입에 적용되는 규칙들에 대해서는 `calls` 절을 보세요. 문법적으로, 어떤 표현식이건 인자로 사용될 수 있습니다; 구해진 값이 지역 변수에 대입됩니다.

용어집의 [매개변수](#) 항목과 FAQ 질문 인자와 매개변수의 차이와 [PEP 362](#)도 보세요.

### asynchronous context manager (비동기 컨텍스트 관리자)

`__aenter__()`와 `__aexit__()` 메서드를 정의함으로써 `async with` 문에서 보이는 환경을 제어하는 객체. [PEP 492](#)로 도입되었습니다.

### asynchronous generator (비동기 제너레이터)

비동기 제너레이터 `이터레이터`를 돌려주는 함수. `async def`로 정의되는 코루틴 함수처럼 보이는데, `async for` 루프가 사용할 수 있는 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다.

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

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

### asynchronous generator iterator (비동기 제너레이터 이터레이터)

비동기 제너레이터 함수가 만드는 객체.

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

각 `yield`는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 `try`-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 `이터레이터`가 `__anext__()`가 돌려주는 또 하나의 어웨이터블로 재개되면, 떠난 곳으로 복귀합니다. [PEP 492](#)와 [PEP 525](#)를 보세요.

### asynchronous iterable (비동기 이터러블)

`async for` 문에서 사용될 수 있는 객체. `__aiter__()` 메서드는 비동기 `이터레이터`를 돌려줘야 합니다. [PEP 492](#)로 도입되었습니다.

### asynchronous iterator (비동기 이터레이터)

`__aiter__()`와 `__anext__()` 메서드를 구현하는 객체. `__anext__()`는 어웨이터블 객체를 돌려줘야 합니다. `async for`는 `StopAsyncIteration` 예외가 발생할 때까지 비동기 `이터레이터`의 `__anext__()` 메서드가 돌려주는 어웨이터블을 푼다. [PEP 492](#)로 도입되었습니다.

### attribute (어트리뷰트)

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

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

### awaitable (어웨이터블)

`await` 표현식에 사용할 수 있는 객체. `코루틴`이나 `__await__()` 메서드를 가진 객체가 될 수 있습니다. [PEP 492](#)를 보세요.

### BDFL

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

### binary file (바이너리 파일)

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



`str` 객체를 읽고 쓸 수 있는 파일 객체에 대해서는 [텍스트 파일](#) 도 참조하세요.

### borrowed reference (빌린 참조)

In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

### bytes-like object (바이트열류 객체)

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

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

### bytecode (바이트 코드)

파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현입니다. 바이트 코드는 `.pyc` 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만듭니다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있습니다). 이 “중간 언어”는 각 바이트 코드에 대응하는 기계를 실행하는 *가상 기계*에서 실행된다고 말합니다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 합니다.

바이트 코드 명령어들의 목록은 `dis` 모듈 설명서에 나옵니다.

### callable (콜러블)

A callable is an object that can be called, possibly with a set of arguments (see [argument](#)), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

### callback (콜백)

인자로 전달되는 미래의 어느 시점에서 실행될 서브 루틴 함수.

### class (클래스)

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

### class variable (클래스 변수)

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

### closure variable (클로저 변수)

A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
x += 1
print(x)
return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

### complex number (복소수)

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

### context (컨텍스트)

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

### context management protocol (컨텍스트 관리 프로토콜)

The `__enter__()` and `__exit__()` methods called by the `with` statement. See [PEP 343](#).

### context manager (컨텍스트 관리자)

컨텍스트 관리 프로토콜을 구현하고 `with` 문에서 보이는 환경을 제어하는 객체. [PEP 343](#)을 참조하십시오.

### context variable (컨텍스트 변수)

A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

### contiguous (연속)

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

### coroutine (코루틴)

코루틴은 서브루틴의 더 일반화된 형태입니다. 서브루틴은 한 지점에서 진입하고 다른 지점에서 탈출합니다. 코루틴은 여러 다른 지점에서 진입하고, 탈출하고, 재개할 수 있습니다. 이것들은 `async def` 문으로 구현할 수 있습니다. [PEP 492](#)를 보세요.

### coroutine function (코루틴 함수)

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

### CPython

파이썬 프로그래밍 언어의 규범적인 구현인데, [python.org](#)에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 “CPython” 이 사용됩니다.

### current context (현재 컨텍스트)

The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

**decorator (데코레이터)**

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

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

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

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

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의와 클래스 정의의 설명서를 보면 됩니다.

**descriptor (디스크립터)**

메서드 `__get__()` 이나 `__set__()` 이나 `__delete__()` 를 정의하는 객체. 클래스 어트리뷰트가 디스크립터일 때, 어트리뷰트 조회는 특별한 연결 작용을 일으킵니다. 보통, *a.b*를 읽거나, 쓰거나, 삭제하는데 사용할 때, *a*의 클래스 디렉터리에서 *b*라고 이름 붙여진 객체를 찾습니다. 하지만 *b*가 디스크립터면, 해당하는 디스크립터 메서드가 호출됩니다. 디스크립터를 이해하는 것은 파이썬에 대한 깊은 이해의 열쇠인데, 함수, 메서드, 프로퍼티, 클래스 메서드, 스태틱 메서드, 슈퍼클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 `descriptors`나 디스크립터 사용법 안내서에 나옵니다.

**dictionary (딕셔너리)**

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

**dictionary comprehension (딕셔너리 컴프리헨션)**

이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 딕셔너리를 반환하는 간결한 방법. `results = {n: n ** 2 for n in range(10)}`은 값 *n* \*\* 2에 매핑된 키 *n*을 포함하는 딕셔너리를 생성합니다. comprehensions을 참조하십시오.

**dictionary view (딕셔너리 뷰)**

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

**docstring (독스트링)**

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

**duck-typing (덕 타이핑)**

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

**EAFP**

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

**expression (표현식)**

어떤 값으로 구해질 수 있는 문법적인 조각. 다른 말로 표현하면, 표현식은 리터럴, 이름, 어트리뷰트

액세스, 연산자, 함수들과 같은 값을 돌려주는 표현 요소들을 쌓아 올린 것입니다. 다른 많은 언어와 대조적으로, 모든 언어 구성물들이 표현식인 것은 아닙니다. `while` 처럼, 표현식으로 사용할 수 없는 문장들이 있습니다. 대입 또한 문장이고, 표현식이 아닙니다.

#### extension module (확장 모듈)

C 나 C++로 작성된 모듈인데, 파이썬의 C API를 사용해서 핵심이나 사용자 코드와 상호 작용합니다.

#### f-string (f-문자열)

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

#### file object (파일 객체)

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

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

#### file-like object (파일류 객체)

파일 객체 의 비슷한 말.

#### filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기)

Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

[로케일 인코딩](#) 도 보세요.

#### finder (파인더)

임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

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

더 자세한 내용은 `finders-and-loaders` 와 `importlib`를 참조하십시오.

#### floor division (정수 나눗셈)

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

#### free threading (자유 스레딩)

A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See [PEP 703](#).

#### free variable (자유 변수)

Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See [closure variable](#) for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

#### function (함수)

호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 인자가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. [매개변수](#) 와 [메서드](#) 와 [function](#) 섹션도 보세요.

**function annotation (함수 어노테이션)**

함수 매개변수나 반환 값의 어노테이션.

함수 어노테이션은 일반적으로 **형 힌트**로 사용됩니다: 예를 들어, 이 함수는 두 개의 `int` 인자를 받아들일 것으로 기대되고, 동시에 `int` 반환 값을 줄 것으로 기대됩니다:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

함수 어노테이션 문법은 **function 절**에서 설명합니다.

이 기능을 설명하는 **변수 어노테이션**과 **PEP 484**를 참조하세요. 또한 어노테이션에 대한 모범 사례는 **annotations-howto**를 참조하세요.

**`__future__`**

A future statement, from `__future__` import <feature>, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

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

**garbage collection (가비지 수거)**

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

**generator (제너레이터)**

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

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

**generator iterator (제너레이터 이터레이터)**

제너레이터 함수가 만드는 객체.

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

**generator expression (제너레이터 표현식)**

**이터레이터**를 돌려주는 **표현식**. 루프 변수와 범위를 정의하는 `for` 절과 생략 가능한 `if` 절이 뒤에 붙는 일반 표현식처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

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

**generic function (제네릭 함수)**

같은 연산을 서로 다른 형들에 대해 구현한 여러 함수로 구성된 함수. 호출 때 어떤 구현이 사용될지는 **디스패치 알고리즘**에 의해 결정됩니다.

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

**generic type (제네릭 형)**

매개 변수화 할 수 있는 **형**; 일반적으로 `list` 나 `dict`와 같은 컨테이너 클래스. **형 힌트**와 **어노테이션**에 사용됩니다.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

**GIL**

전역 **인터프리터** **락**을 보세요.



**global interpreter lock (전역 인터프리터 록)**

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

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

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see [PEP 703](#).

**hash-based pyc (해시 기반 pyc)**

유효성을 판별하기 위해 해당 소스 파일의 최종 수정 시간이 아닌 해시를 사용하는 바이트 코드 캐시 파일. `pyc-invalidation`을 참조하세요.

**hashable (해시 가능)**

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

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

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

**IDLE**

파이썬을 위한 통합 개발 및 학습 환경 (Integrated Development and Learning Environment). `idle`은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

**immortal (불멸)**

*Immortal objects* are a CPython implementation detail introduced in [PEP 683](#).

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

**immutable (불변)**

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

**import path (임포트 경로)**

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

**importing (임포트)**

한 모듈의 파이썬 코드가 다른 모듈의 파이썬 코드에서 사용될 수 있도록 하는 절차.

**importer (임포터)**

모듈을 찾기도 하고 로드 하기도 하는 객체; 동시에 **파인더**이자 **로더** 객체입니다.

**interactive (대화형)**

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

**interpreted (인터프리터드)**

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

**interpreter shutdown (인터프리터 종료)**

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

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

**iterable (이터러블)**

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

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

**iterator (이터레이터)**

데이터의 스트림을 표현하는 객체. 이터레이터의 `__next__()` 메서드를 반복적으로 호출하면 (또는 내장 함수 `next()` 로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 `StopIteration` 예외를 일으킵니다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 `__next__()` 메서드 호출은 `StopIteration` 예외를 다시 일으키기만 합니다. 이터레이터는 이터레이터 객체 자신을 돌려주는 `__iter__()` 메서드를 가질 것이 요구되기 때문에, 이터레이터는 이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션 시도하는 코드입니다. (`list` 같은) 컨테이너 객체는 `iter()` 함수로 전달하거나 `for` 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

`typeiter` 에 더 자세한 내용이 있습니다.

**CPython 구현 상세:** CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

**key function (키 함수)**

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

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

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

**keyword argument (키워드 인자)**

[인자](#) 를 보세요.

**lambda (람다)**

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

**LBYL**

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

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

**lexical analyzer (어휘 분석기)**

Formal name for the *tokenizer*; see *token*.

**list (리스트)**

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

**list comprehension (리스트 컴프리헨션)**

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

**loader (로더)**

An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also:

- `finders-and-loaders`
- `importlib.abc.Loader`
- **PEP 302**

**locale encoding (로케일 인코딩)**

On Unix, it is the encoding of the `LC_CTYPE` locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: "cp1252").

On Android and VxWorks, Python uses "utf-8" as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

**magic method (매직 메서드)**

특수 메서드 의 비공식적인 비슷한 말.

**mapping (매핑)**

임의의 키 조회를 지원하고 `collections.abc.Mapping` 이나 `collections.abc.MutableMapping` 추상 베이스 클래스 에 지정된 메서드들을 구현하는 컨테이너 객체. 예로는 `dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` 를 들 수 있습니다.

**meta path finder (메타 경로 파인더)**

`sys.meta_path` 의 검색이 돌려주는 **파인더**. 메타 경로 파인더는 **경로 엔트리 파인더** 와 관련되어 있기는 하지만 다릅니다.

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

**metaclass (메타 클래스)**

클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 디렉터리, 베이스 클래스들의 목록을 만듭니다. 메타 클래스는 이 세 인자를 받아서 클래스를 만드는 책임을 집니다. 대부분의 객체 지향형 프로그래밍 언어들은 기본 구현을 제공합니다. 파이썬을 특별하게 만드는 것은 커스텀 메타클래스를 만들 수 있다는 것입니다. 대부분 사용자에게는 이 도구가 전혀 필요 없지만, 필요가 생길 때, 메타 클래스는



강력하고 우아한 해법을 제공합니다. 어트리뷰트 액세스의 로깅(logging), 스레드 안전성의 추가, 객체 생성 추적, 싱글톤 구현과 많은 다른 작업에 사용됐습니다.

metaclasses 에서 더 자세한 내용을 찾을 수 있습니다.

### method (메서드)

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

### method resolution order (메서드 결정 순서)

메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서입니다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 `python_2.3_mro` 를 보세요.

### module (모듈)

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

**패키지** 도 보세요.

### module spec (모듈 스펙)

모듈을 로드하는데 사용되는 임포트 관련 정보들을 담고 있는 이름 공간. `importlib.machinery.ModuleSpec` 의 인스턴스.

`module-specs` 도 보세요.

### MRO

**메서드 결정 순서** 를 보세요.

### mutable (가변)

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

### named tuple (네임드 튜플)

“named tuple(네임드 튜플)”이라는 용어는 튜플에서 상속하고 이름 붙은 어트리뷰트를 사용하여 인덱스 할 수 있는 요소에 액세스 할 수 있는 모든 형이나 클래스에 적용됩니다. 형이나 클래스에는 다른 기능도 있을 수 있습니다.

`time.localtime()` 과 `os.stat()` 가 반환한 값을 포함하여, 여러 내장형이 네임드 튜플입니다. 또 다른 예는 `sys.float_info`입니다:

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

일부 네임드 튜플은 내장형(위의 예)입니다. 또는, `tuple`에서 상속하고 이름 붙은 필드를 정의하는 일반 클래스 정의로 네임드 튜플을 만들 수 있습니다. 이러한 클래스는 직접 작성하거나, `typing.NamedTuple`를 계승하거나 팩토리 함수 `collections.namedtuple()` 로 만들 수 있습니다. 후자의 기법은 직접 작성하거나 내장 네임드 튜플에서는 찾을 수 없는 몇 가지 추가 메서드를 추가하기도 합니다.

### namespace (이름 공간)

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

### namespace package (이름 공간 패키지)

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

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see [PEP 420](#) and [reference-namespace-package](#).

모듈도 보세요.

#### nested scope (중첩된 스코프)

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

#### new-style class (뉴스타일 클래스)

지금은 모든 클래스 객체에 사용되고 있는 클래스 버전의 예전 이름. 초기의 파이썬 버전에서는, 오직 뉴스타일 클래스만 `__slots__`, 디스크립터, 프라퍼티, `__getattr__()`, 클래스 메서드, 스태틱 메서드와 같은 파이썬의 새롭고 다양한 기능들을 사용할 수 있었습니다.

#### object (객체)

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

#### optimized scope (최적화된 스코프)

A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

#### package (패키지)

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

정규 패키지 와 이름 공간 패키지 도 보세요.

#### parameter (매개변수)

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

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

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

- 위치-전용 (*positional-only*): 위치로만 제공될 수 있는 인자를 지정합니다. 위치 전용 매개변수는 함수 정의의 매개변수 목록에 / 문자를 포함하고 그 뒤에 정의할 수 있습니다, 예를 들어 다음에서 `posonly1` 과 `posonly2`:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- 키워드-전용 (*keyword-only*): 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 \*를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 `kw_only1` 와 `kw_only2`:

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

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

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

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

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

인자 용어집 항목, 인자와 매개변수의 차이에 나오는 FAQ 질문, `inspect.Parameter` 클래스, `function` 절, [PEP 362](#)도 보세요.

### path entry (경로 엔트리)

경로 기반 파인더가 임포트 할 모듈들을 찾기 위해 참고하는 임포트 경로 상의 하나의 장소.

### path entry finder (경로 엔트리 파인더)

`sys.path_hooks`에 있는 콜러블 (즉, 경로 엔트리 [혹](#)) 이 돌려주는 파인더 인데, 주어진 경로 엔트리로 모듈을 찾는 방법을 알고 있습니다.

경로 엔트리 파인더들이 구현하는 메서드들은 `importlib.abc.PathEntryFinder`에 나옵니다.

### path entry hook (경로 엔트리 [혹](#))

`sys.path_hooks` 리스트에 있는 콜러블인데, 특정 경로 엔트리에서 모듈을 찾는 법을 알고 있다면 경로 엔트리 파인더를 돌려줍니다.

### path based finder (경로 기반 파인더)

기본 메타 경로 파인더들 중 하나인데, 임포트 경로에서 모듈을 찾습니다.

### path-like object (경로류 객체)

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

### PEP

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

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

[PEP 1](#) 참조하세요.

### portion (포션)

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

### positional argument (위치 인자)

인자를 보세요.

### provisional API (잠정 API)

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

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

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

### provisional package (잠정 패키지)

잠정 API를 보세요.

**Python 3000 (파이썬 3000)**

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

**Pythonic (파이썬다운)**

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

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

더 깔끔한, 파이썬다운 방법은 이렇습니다:

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

**qualified name (정규화된 이름)**

모듈의 전역 스코프에서 모듈에 정의된 클래스, 함수, 메서드에 이르는 “경로”를 보여주는 점으로 구분된 이름. **PEP 3155** 에서 정의됩니다. 최상위 함수와 클래스의 경우에, 정규화된 이름은 객체의 이름과 같습니다:

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

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

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

**reference count (참조 횟수)**

객체에 대한 참조의 개수. 객체의 참조 횟수가 0으로 떨어지면, 메모리가 반납됩니다. 일부 객체는 불멸이며 참조 횟수가 수정되지 않아서, 객체가 할당 해제되지 않습니다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소입니다. 프로그래머는 특정 객체의 참조 횟수를 돌려주는 `sys.getrefcount()` 함수를 호출할 수 있습니다.

**regular package (정규 패키지)**

`__init__.py` 파일을 포함하는 디렉터리와 같은 전통적인 패키지.

이름 공간 패키지 도 보세요.

**REPL**

An acronym for the “read-eval-print loop”, another name for the *interactive* interpreter shell.

**\_\_slots\_\_**

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

**sequence (시퀀스)**

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

`collections.abc.Sequence` 추상 베이스 클래스는 `__getitem__()` 과 `__len__()` 을 넘어서 훨씬 풍부한 인터페이스를 정의하는데, `count()`, `index()`, `__contains__()`, `__reversed__()` 를 추가합니다. 이 확장된 인터페이스를 구현한 형을 `register()` 를 사용해서 명시적으로 등록할 수 있습니다. 시퀀스 메서드 일반에 대한 자세한 문서는, 공통 시퀀스 연산을 참조하세요.

**set comprehension (집합 컴프리헨션)**

이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 집합을 반환하는 간결한 방법. `results = {c for c in 'abracadabra' if c not in 'abc'}` 는 문자열의 집합 `{'r', 'd'}` 를 생성합니다. `comprehensions` 을 참조하십시오.

**single dispatch (싱글 디스패치)**

구현이 하나의 인자의 형에 기초해서 결정되는 **제네릭 함수** 디스패치의 한 형태.

**slice (슬라이스)**

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

**soft deprecated (약하게 폐지된)**

A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See [PEP 387: Soft Deprecation](#).

**special method (특수 메서드)**

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

**statement (문장)**

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

**static type checker (정적 형 검사기)**

An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also [type hints](#) and the `typing` module.

**strong reference (강한 참조)**

In Python’s C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

[빌린 참조](#) 도 보세요.

**text encoding (텍스트 인코딩)**

A string in Python is a sequence of Unicode code points (in range U+0000–U+10FFFF). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as “encoding”, and recreating the string from the sequence of bytes is known as “decoding”.

There are a variety of different text serialization codecs, which are collectively referred to as “text encodings”.



**text file (텍스트 파일)**

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

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

**token (토큰)**

A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python's lexical analyzer. The `token` module contains information on the various types of tokens.

**triple-quoted string (삼중 따옴표 된 문자열)**

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

**type (형)**

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

**type alias (형 에일리어스)**

형을 식별자에 대입하여 만들어지는 형의 동의어.

형 에일리어스는 **형 힌트**를 단순화하는 데 유용합니다. 예를 들면:

```
def remove_gray_shades(
    colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
Color = tuple[int, int, int]

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

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

**type hint (형 힌트)**

변수, 클래스 어트리뷰트 및 함수 매개변수 나 반환 값의 기대되는 형을 지정하는 **어노테이션**.

형 힌트는 선택 사항이고 파이썬에서 강제되지는 않지만, **정적 형 검사기**에 유용합니다. 또한 IDE의 코드 완성 및 리팩토링을 돕습니다.

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

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

**universal newlines (유니버설 줄 넘김)**

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

**variable annotation (변수 어노테이션)**

변수 또는 클래스 어트리뷰트의 **어노테이션**.

변수 또는 클래스 어트리뷰트에 어노테이션을 달 때 대입은 선택 사항입니다:

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

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

```
count: int = 0
```

변수 어노테이션 문법은 섹션 [annassign](#)에서 설명합니다.

이 기능을 설명하는 [함수 어노테이션](#), [PEP 484](#) 및 [PEP 526](#)을 참조하세요. 또한 어노테이션 작업에 대한 모범 사례는 [annotations-howto](#)를 참조하세요.

### virtual environment (가상 환경)

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

`venv`도 보세요.

### virtual machine (가상 기계)

소프트웨어만으로 정의된 컴퓨터. 파이썬의 가상 기계는 바이트 코드 컴파일러가 출력하는 **바이트 코드**를 실행합니다.

### Zen of Python (파이썬 젠)

파이썬 디자인 원리와 철학들의 목록인데, 언어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 `import this`를 입력하면 보입니다.





---

### 이 설명서에 관하여

---

파이썬 설명서는 `reStructuredText` 소스에서 만들어진 것으로, 원래 파이썬을 위해 제작되었고 이제는 독립 프로젝트로 유지 관리되는 설명서 생성기인 `Sphinx` 를 사용했습니다.

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

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 저자;
- `reStructuredText`와 `Docutils` 스위트를 만드는 `Docutils` 프로젝트.
- Fredrik Lundh, 그의 대안 파이썬 참조(Alternative Python Reference) 프로젝트에서 `Sphinx`가 많은 아이디어를 얻었습니다.

### B.1 파이썬 설명서의 공헌자들

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

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



## 역사와 라이선스

## C.1 소프트웨어의 역사

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <https://www.cwi.nl>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <https://www.cnri.reston.va.us>) in Reston, Virginia where he released several versions of the software.

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

All Python releases are Open Source (see <https://opensource.org> for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

배포판	파생된 곳	해	소유자	GPL-compatible? (1)
0.9.0 ~ 1.2	n/a	1991-1995	CWI	yes
1.3 ~ 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 이상	2.1.1	2001-현재	PSF	yes

## 참고

- (1) GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-

compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

- (2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is "not incompatible" with the GPL.

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

## C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

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

### C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using this software ("Python") in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001–2024 Python Software Foundation; All Rights Reserved" are retained in Python alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to ↪Python.
4. PSF is making Python available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License

(다음 페이지에 계속)

(이전 페이지에서 계속)

Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

### BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and

(다음 페이지에 계속)

(이전 페이지에서 계속)

- otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: <http://hdl.handle.net/1895.22/1013>".
  3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
  4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
  5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
  6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
  7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
  8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

## C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

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

## C.3 포함된 소프트웨어에 대한 라이선스 및 승인

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

### C.3.1 메르센 트위스터

The `_random` C extension underlying the `random` module includes code based on a download from <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. The following are the verbatim comments from the original code:

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

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

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

Redistribution and use in source and binary forms, with or without
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

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

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

### C.3.2 소켓

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, <https://www.wide.ad.jp/>.

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

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

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

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL

(다음 페이지에 계속)



(이전 페이지에서 계속)

DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.3 비동기 소켓 서비스

The `test.support.asyncchat` and `test.support.asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

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

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

### C.3.4 쿠키 관리

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

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

All Rights Reserved

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.5 실행 추적

trace 모듈은 다음과 같은 주의 사항을 포함합니다:

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

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

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

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

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

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

### C.3.6 UUencode 및 UUdecode 함수

The uu codec contains the following notice:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

Modified by Jack Jansen, CWI, July 1995:

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

### C.3.7 XML 원격 프로시저 호출

xmlrpc.client 모듈은 다음과 같은 주의 사항을 포함합니다:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB

Copyright (c) 1999-2002 by Fredrik Lundh

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

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

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

### C.3.8 test\_epoll

The test.test\_epoll module contains the following notice:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

### C.3.9 Select kqueue

select 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

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

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

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

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

### C.3.10 SipHash24

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

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

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

Solution inspired by code from:  
 Samuel Neves (supercop/crypto\_auth/siphash24/little)  
 djb (supercop/crypto\_auth/siphash24/little2)  
 Jean-Philippe Aumasson (<https://131002.net/siphash/siphash24.c>)

### C.3.11 strtod 와 dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```

/*****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 *****/

```

### C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```

                        Apache License
                        Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction,
and distribution as defined by Sections 1 through 9 of this document.

"Licenser" shall mean the copyright owner or entity authorized by
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual,

(다음 페이지에 계속)

(이전 페이지에서 계속)

worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
  - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
  - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
  - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
  - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,

(다음 페이지에 계속)

(이전 페이지에서 계속)

any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

### C.3.13 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

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

Permission is hereby granted, free of charge, to any person obtaining

(다음 페이지에 계속)



(이전 페이지에서 계속)

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

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

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

### C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

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

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

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

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

### C.3.15 zlib

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly  
jloup@gzip.org

Mark Adler  
madler@alumni.caltech.edu

### C.3.16 cfuhash

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

Copyright (c) 2005 Don Owens  
All rights reserved.

This code is released under the BSD license:

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

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

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

### C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the `libmpdec` library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.
```

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

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

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

### C.3.18 W3C C14N 테스트 스위트

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at <https://www.w3.org/TR/xml-c14n2-testcases/> and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),  
All Rights Reserved.
```

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

- \* Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

### C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen
```

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

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

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

### C.3.20 asyncio

Parts of the `asyncio` module are incorporated from `uvloop 0.16`, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc. http://magic.io
```

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

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

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

### C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in `subr_smr.c`. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>
```

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

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

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



## APPENDIX D

---

### 저작권

---

파이썬과 이 설명서는:

Copyright © 2001-2024 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

---

전체 라이선스 및 사용 권한 정보는 [역사와 라이선스](#) 에서 제공합니다.





## 알파벳 이외

..., [121](#)  
 # (*hash*)  
     comment, [9](#)  
 \* (*asterisk*)  
     in function calls, [32](#)  
 \*\*  
     in function calls, [32](#)  
 : (*colon*)  
     function annotations, [34](#)  
 ->  
     function annotations, [34](#)  
 >>>, [121](#)  
 \_\_all\_\_, [54](#)  
 \_\_future\_\_, [127](#)  
 \_\_slots\_\_, [134](#)

## A

abstract base class (추상 베이스 클래스), [121](#)  
 annotation (어노테이션), [121](#)  
 annotations  
     function, [34](#)  
 argument (인자), [121](#)  
 asynchronous context manager (비동기 컨텍스트 관리자), [122](#)  
 asynchronous generator (비동기 제너레이터), [122](#)  
 asynchronous generator iterator (비동기 제너레이터 이터레이터), [122](#)  
 asynchronous iterable (비동기 이터러블), [122](#)  
 asynchronous iterator (비동기 이터레이터), [122](#)  
 attribute (어트리뷰트), [122](#)  
 awaitable (어웨이터블), [122](#)

## B

BDFL, [122](#)  
 binary file (바이너리 파일), [122](#)  
 borrowed reference (빌린 참조), [123](#)  
 built-in function  
     help, [91](#)  
     open, [61](#)  
 builtins

module, [51](#)

bytecode (바이트 코드), [123](#)  
 bytes-like object (바이트열류 객체), [123](#)

## C

callable (콜러블), [123](#)  
 callback (콜백), [123](#)  
 C-contiguous, [124](#)  
 class (클래스), [123](#)  
 class variable (클래스 변수), [123](#)  
 closure variable (클로저 변수), [123](#)  
 coding  
     style, [34](#)  
 complex number (복소수), [124](#)  
 context (컨텍스트), [124](#)  
 context management protocol (컨텍스트 관리 프로토콜), [124](#)  
 context manager (컨텍스트 관리자), [124](#)  
 context variable (컨텍스트 변수), [124](#)  
 contiguous (연속), [124](#)  
 coroutine (코루틴), [124](#)  
 coroutine function (코루틴 함수), [124](#)  
 CPython, [124](#)  
 current context (현재 컨텍스트), [124](#)

## D

decorator (데코레이터), [125](#)  
 descriptor (디스크립터), [125](#)  
 dictionary (딕셔너리), [125](#)  
 dictionary comprehension (딕셔너리 컴프리헨션), [125](#)  
 dictionary view (딕셔너리 뷰), [125](#)  
 docstring (독스트링), [125](#)  
 docstrings, [26](#), [33](#)  
 documentation strings, [26](#), [33](#)  
 duck-typing (덕 타이핑), [125](#)

## E

EAFP, [125](#)  
 expression (표현식), [125](#)  
 extension module (확장 모듈), [126](#)

## F

f-string (f-문자열), [126](#)

- file
  - object, 61
- file object (파일 객체), 126
- file-like object (파일류 객체), 126
- filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기), 126
- finder (파인더), 126
- floor division (정수 나눗셈), 126
- for
  - statement, 19
- formatted string literal, 58
- Fortran contiguous, 124
- free threading (자유 스레딩), 126
- free variable (자유 변수), 126
- fstring, 58
- f-string, 58
- function
  - annotations, 34
- function (함수), 126
- function annotation (함수 어노테이션), 127

## G

- garbage collection (가비지 수거), 127
- generator (제너레이터), 127
- generator expression (제너레이터 표현식), 127
- generator iterator (제너레이터 이터레이터), 127
- generic function (제네릭 함수), 127
- generic type (제네릭 형), 127
- GIL, 127
- global interpreter lock (전역 인터프리터 록), 128

## H

- hash-based pyc (해시 기반 *pyc*), 128
- hashable (해시 가능), 128
- help
  - built-in function, 91

## I

- IDLE, 128
- immortal (불멸), 128
- immutable (불변), 128
- import path (임포트 경로), 128
- importer (임포터), 128
- importing (임포팅), 128
- interactive (대화형), 128
- interpolated string literal, 58
- interpreted (인터프리티드), 129
- interpreter shutdown (인터프리터 종료), 129
- iterable (이터러블), 129
- iterator (이터레이터), 129

## J

- json
  - module, 63

## K

- key function (키 함수), 129
- keyword argument (키워드 인자), 129

## L

- lambda (람다), 130
- LBYL, 130
- lexical analyzer (어휘 분석기), 130
- list (리스트), 130
- list comprehension (리스트 컴프리헨션), 130
- loader (로더), 130
- locale encoding (로케일 인코딩), 130

## M

- magic
  - method (메서드), 130
- magic method (매직 메서드), 130
- mangling
  - name, 86
- mapping (매핑), 130
- meta path finder (메타 경로 파인더), 130
- metaclass (메타 클래스), 130
- method
  - object, 81
- method (메서드), 131
  - magic, 130
  - special, 135
- method resolution order (메서드 결정 순서), 131
- module
  - builtins, 51
  - json, 63
  - search path, 49
  - sys, 50
- module (모듈), 131
- module spec (모듈 스펙), 131
- MRO, 131
- mutable (가변), 131

## N

- name
  - mangling, 86
- named tuple (네임드 튜플), 131
- namespace (이름 공간), 131
- namespace package (이름 공간 패키지), 131
- nested scope (중첩된 스코프), 132
- new-style class (뉴스타일 클래스), 132

## O

- object
  - file, 61
  - method, 81
- object (객체), 132
- open
  - built-in function, 61
- optimized scope (최적화된 스코프), 132

## P

package (패키지), [132](#)  
 parameter (매개변수), [132](#)  
 PATH, [49](#), [120](#)  
 path  
   module search, [49](#)  
 path based finder (경로 기반 파인더), [133](#)  
 path entry (경로 엔트리), [133](#)  
 path entry finder (경로 엔트리 파인더), [133](#)  
 path entry hook (경로 엔트리 훅), [133](#)  
 path-like object (경로류 객체), [133](#)  
 PEP, [133](#)  
 portion (포션), [133](#)  
 positional argument (위치 인자), [133](#)  
 provisional API (잠정 API), [133](#)  
 provisional package (잠정 패키지), [133](#)  
 Python 3000 (파이썬 3000), [134](#)  
 Python **향상 제안**  
   PEP 1, [133](#)  
   PEP 8, [34](#)  
   PEP 238, [126](#)  
   PEP 278, [136](#)  
   PEP 302, [130](#)  
   PEP 343, [124](#)  
   PEP 362, [122](#), [133](#)  
   PEP 411, [133](#)  
   PEP 420, [132](#), [133](#)  
   PEP 443, [127](#)  
   PEP 483, [127](#)  
   PEP 484, [34](#), [121](#), [127](#), [136](#), [137](#)  
   PEP 492, [122](#), [124](#)  
   PEP 498, [126](#)  
   PEP 519, [133](#)  
   PEP 525, [122](#)  
   PEP 526, [121](#), [137](#)  
   PEP 585, [127](#)  
   PEP 636, [25](#)  
   PEP 683, [128](#)  
   PEP 703, [126](#), [128](#)  
   PEP 3107, [34](#)  
   PEP 3116, [136](#)  
   PEP 3147, [50](#)  
   PEP 3155, [134](#)  
 PYTHON\_BASIC\_REPL, [119](#)  
 PYTHON\_GIL, [128](#)  
 Pythonic (파이썬다운), [134](#)  
 PYTHONPATH, [49](#), [51](#)  
 PYTHONSTARTUP, [120](#)

## Q

qualified name (정규화된 이름), [134](#)

## R

reference count (참조 횟수), [134](#)  
 regular package (정규 패키지), [134](#)  
 REPL, [134](#)  
 RFC  
   RFC 2822, [95](#)

## S

search  
   path, module, [49](#)  
 sequence (시퀀스), [135](#)  
 set comprehension (집합 컴프리헨션), [135](#)  
 single dispatch (싱글 디스패치), [135](#)  
 sitecustomize, [120](#)  
 slice (슬라이스), [135](#)  
 soft deprecated (약하게 폐지된), [135](#)  
 special  
   method (메서드), [135](#)  
 special method (특수 메서드), [135](#)  
 statement  
   for, [19](#)  
 statement (문장), [135](#)  
 static type checker (정적 형 검사기), [135](#)  
 string  
   formatted literal, [58](#)  
   interpolated literal, [58](#)  
 strings, documentation, [26](#), [33](#)  
 strong reference (강한 참조), [135](#)  
 style  
   coding, [34](#)  
 sys  
   module, [50](#)

## T

text encoding (텍스트 인코딩), [135](#)  
 text file (텍스트 파일), [136](#)  
 token (토큰), [136](#)  
 triple-quoted string (삼중 따옴표 된 문자열), [136](#)  
 type (형), [136](#)  
 type alias (형 에일리어스), [136](#)  
 type hint (형 힌트), [136](#)

## U

universal newlines (유니버설 줄 넘김), [136](#)  
 usercustomize, [120](#)

## V

variable annotation (변수 어노테이션), [136](#)  
 virtual environment (가상 환경), [137](#)  
 virtual machine (가상 기계), [137](#)

## Y

**환경 변수**  
   PATH, [49](#), [120](#)  
   PYTHON\_BASIC\_REPL, [119](#)  
   PYTHON\_GIL, [128](#)  
   PYTHONPATH, [49](#), [51](#)  
   PYTHONSTARTUP, [120](#)

## Z

Zen of Python (파이썬 젠), [137](#)