Python Tutorial

출시 버전 *3.6.15*

Guido van Rossum
and the Python development team

9월 05, 2021

Contents

파이썬은 배우기 쉽고, 강력한 프로그래밍 언어입니다. 효율적인 자료 구조들과 객체 지향 프로그래밍에 대해 간단하고도 효과적인 접근법을 제공합니다. 우아한 문법과 동적 타이핑(typing)은, 인터프리터 적인 특징들과 더불어, 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.

CHAPTER 1

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 적업들을 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 만드는데도 막대한 개발 시간이 들어갑니다. 파이썬은 사용하기에 더 간단하고, 윈도우, 맥 OS X, 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 에러 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.

파이썬 이라는 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.

CHAPTER 2

파이썬 인터프리터 사용하기

2.1 인터프리터 실행하기

파이썬 인터프리터는 보통 `/usr/local/bin/python3.6` 에 설치됩니다; 유닉스 셸의 검색 경로에 `/usr/local/bin` 를 넣으면 명령:

```
python3.6
```

을 셸에 입력해서 실행할 수 있습니다.[1] 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.)

윈도우에서, 파이썬 설치는 보통 `C:\Python36` 에 이루어지지만, 설치기를 실행할 때 변경할 수 있습니다. 이 디렉터리를 경로에 넣으려면 다음과 같은 명령을 DOS 상자의 명령 프롬프트에 입력하면 됩니다:

```
set path=%path%;C:\python36
```

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 readline 을 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. 삑 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.

[1] 유닉스에서, 파이썬 3.x 인터프리터는 보통 `python` 이라는 이름의 실행 파일로 설치되지 않는데, 동시에 설치되는 파이썬 2.x 실행 파일과 충돌하지 않도록 하기 위해섭니다.

인터프리터를 실행하는 두 번째 방법은 `python -c command [arg] ...` 인데, *command* 에 있는 문장들을 실행합니다. 셸의 `-c` 옵션에 해당합니다. 파이썬 문장은 종종 셸에서 특별한 의미가 있는 공백이나 다른 문자들을 포함하기 때문에, *command* 전체를 작은따옴표로 감싸주는 것이 좋습니다.

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.

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 1이고, 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 모드 로 동작한다고 말합니다. 이 모드에서는 기본 프롬프트 를 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.

```
$ python3.6
Python 3.6 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] 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 인코드 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 올바로 표시하기 위해서는 편집기가 파일이 UTF-8임을 인식해야 하고, 이 파일에 포함된 모든 문자를 지원할 수 있는 폰트를 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)*》 줄 로 시작하는 경우입니다. 이 경우에, 인코딩 선언은 두 번째 줄에 들어갑니다. 예를 들어:

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

CHAPTER 3

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

인터프리터는 간단한 계산기로 기능합니다: 표현식을 입력하면 값을 출력합니다. 표현식 문법은 간단합니다. +, -, *, / 연산자들은 대부분의 다른 언어들 (예를 들어, 파스칼이나 C) 처럼 동작합니다; 괄호 (()) 는 묶는 데 사용합니다. 예를 들어:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 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 관해 더 자세히 살펴볼 예정입니다.

나눗셈 (/) 은 항상 float를 돌려줍니다. 정수 나눗셈 으로 (소수부 없이) 정수 결과를 얻으려면 // 연산자를 사용하면 됩니다; 나머지를 얻으려면 % 를 사용할 수 있습니다:

```
>>> 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  # result * divisor + remainder
17
```

파이썬에서는 거듭제곱을 계산할 때 ** 연산자를 사용합니다[1]:

```
>>> 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:

[1] ** 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.

```
>>> 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 + _
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 문자열

숫자와는 별개로, 파이썬은 문자열도 다룰 수 있는데 여러 가지 방법으로 표현됩니다. 작은따옴표('...') 나 큰따옴표("...")로 둘러쌀 수 있는데 모두 같은 결과를 줍니다[2]. 따옴표를 이스케이핑 할 때는 \ 를 사용할 수 있습니다:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> '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.'
```

대화형 인터프리터에서, 출력 문자열은 따옴표로 둘러싸여 있고, 특수 문자들은 역 슬래시로 이스케이핑 됩니다. 때로 입력한 것과 달라 보여도 (따옴표의 종류가 바뀔 수 있다), 두 문자열은 동등합니다. 문자열이 작은따옴표를 포함하고 큰따옴표를 포함하지 않으면 큰따옴표가 사용되고, 그 외의 경우는 작은따옴표가 사용됩니다. print() 함수는 따옴표를 생략하고, 이스케이핑된 특수 문자를 출력해서 더 읽기 쉬운 출력을 만들어냅니다:

[2] 다른 언어들과는 달리, "" 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.

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a 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:

```
>>> 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
```

문자열 리터럴은 여러 줄로 확장될 수 있습니다. 한 가지 방법은 삼중 따옴표를 사용하는 것입니다: `"""..."""` 또는 `'''...'''`. 줄 넘김 문자는 자동으로 문자열에 포함됩니다. 하지만 줄 끝에 \ 를 붙여 이를 방지할 수도 있습니다. 다음 예:

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

는 이런 결과를 출력합니다 (첫 번째 개행문자가 포함되지 않는 것에 주목하세요):

```
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
  ...
SyntaxError: invalid syntax
>>> ('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.

인덱싱에 더해 슬라이싱 *(slicing)* 도 지원됩니다. 인덱싱이 개별 문자를 얻는데 사용되는 반면, 슬라이싱 은 부분 문자열 (substring) 을 얻는 데 사용됩니다:

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

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'
```

슬라이스 인덱스는 편리한 기본값을 갖고 있습니다; 첫 번째 인덱스를 생략하면 기본값 0 이 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'
```

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…6 의 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 차입니다. 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'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

다른 문자열이 필요하면, 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 형 의 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)* 자료 형을 알고 있습니다. 가장 융통성이 있는 것은 리스트 인데, 꺾쇠괄호 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]
```

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

```
>>> squares[:]
[1, 4, 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]
```

`append()` 메서드 *(method)* (나중에 메서드에 대해 더 자세히 알아볼 것입니다) 를 사용하면 리스트의 끝에 새 항목을 추가할 수 있습니다:

```
>>> 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]
```

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

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
['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

물론, 2 에 2를 더하는 것보다는 더 복잡한 방법으로 파이썬을 사용할 수 있습니다. 예를 들어, 다음처럼 피보나치 *(Fibonacci)* 수열의 앞부분을 계산할 수 있습니다:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
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 0과 1이 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: `b < 10`)이 참인 동안 실행됩니다. C와 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 C와 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.
- `print()` 함수는 주어진 인자들의 값을 인쇄합니다. 다중 인자, 실수의 값, 문자열을 다루는 방식에서 (계산기 예제에서 본 것과 같이) 출력하고자 하는 표현식을 그냥 입력하는 것과는 다릅니다. 문자열은 따옴표 없이 출력되고, 인자들 간에는 빈칸이 삽입됩니다. 그래서 이런 식으로 보기 좋게 포매팅할 수 있습니다:

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

CHAPTER 4

기타 제어 흐름 도구

방금 소개한 while 문 외에도, 파이썬은 다른 언어들에서 알려진 일반적인 제어 흐름 문들을 알고 있으며, 나름의 변형을 가하고 있습니다.

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.

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
```

루프 안에서 이터레이트하는 시퀀스를 수정할 필요가 있다면 (예를 들어, 선택한 항목들을 중복시키기), 먼저 사본을 만들 것을 권합니다. 시퀀스를 이터레이트할 때 묵시적으로 사본이 만들어지지는 않습니다. 슬라이스 표기법은 이럴 때 특히 편리합니다:

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

for w in words: 를 쓰면, 위의 예는 defenestrate 를 반복해서 넣고 또 넣음으로써, 무한한 리스트를 만들려고 시도하게 됩니다.

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:

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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)):
...     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:

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

많은 경우에 range() 가 돌려준 객체는 리스트인 것처럼 동작하지만, 사실 리스트가 아닙니다. 이터레이트할 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. 함수 list() 도 그런 것입니다; 이터러블로 리스트를 만듭니다:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

나중에 이터러블을 돌려주고 이터러블을 인자로 받는 함수들을 더 보게 됩니다.

4.4 루프의 break 와 continue 문, 그리고 else 절

break 문은, C처럼, 가장 가까이서 둘러싸는 for 나 while 루프로부터 빠져나가게 만듭니다.

루프 문은 else 절을 가질 수 있습니다; 루프가 리스트의 소진이나 (for 의 경우) 조건이 거짓이 돼서 (while 의 경우) 종료할 때 실행됩니다. 하지만 루프가 break 문으로 종료할 때는 실행되지 않습니다. 소수를 찾는 루프를 통해 다음에서 예시합니다:

```
>>> 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(이것은 올바른 코드입니다. 자세히 들여다보면: else 절은 if 문이 **아니라** for 루프에 속합니다.)

루프와 함께 사용될 때, else 절은 if 문보다는 try 문의 else 절과 비슷한 면이 많습니다: try 문의 else 절은 예외가 발생하지 않을 때 실행되고, 루프의 else 절은 break 가 발생하지 않을 때 실행됩니다. try 문과 예외에 관한 자세한 내용은 예외 처리하기 를 보세요.

continue 문은, 역시 C에서 빌렸습니다, 루프의 다음 이터레이션에서 계속하도록 만듭니다:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5 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 사용된다:

```
>>> 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!
...
```

4.6 함수 정의하기

피보나치 수열을 임의의 한도까지 출력하는 함수를 만들 수 있습니다:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         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 정의 를 시작합니다. 함수 이름과 괄호로 싸인 형식 파라미터들의 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 토큐멘테이션 문자열, 즉 독스트링 *(docstring)* 입니다. (독스트링에 대한 자세한 내용은 도큐멘테이션 문자열 에 나옵니다.) 독스트링을 사용해서 온라인이나 인쇄된 도큐멘테이션을 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 실행 은 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 살핍니다. 그래서, 참조될 수는 있다 하더라도, 전역 변수들은 함수 내에서 (`global` 문으로 명시하지 않는 이상) 직접 값이 대입될 수 없습니다.

함수 호출로 전달되는 실제 파라미터들 (인자들) 은 호출될 때 호출되는 함수의 지역 심볼 테이블에 만들어집니다; 그래서 인자들은 값에 의한 호출 *(call by value)* 로 전달됩니다 (값 은 항상 객체의 값이 아니라 객체 참조입니다).[1] 함수가 다른 함수를 호출할 때, 그 호출을 위한 새 지역 심볼 테이블이 만들어집니다.

함수 정의는 현재 심볼 테이블에 함수 이름을 만듭니다. 함수 이름의 값은 인터프리터가 사용자 정의 함수로 인식하는 형입니다. 이 값은 다른 이름에 대입될 수 있는데, 이 역시 함수로 사용될 수 있습니다. 이것이 이름을 바꾸는 일반적인 방법입니다:

```
>>> 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:

[1] 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).

```
>>> 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
...
>>> 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.
- 문장 result.append(a) 은 리스트 객체 result 의 메서드 를 호출합니다. 메서드는 객체에 〈속하는〉 함수이고 obj.methodname 라고 이름 붙여지는데, obj 는 어떤 객체이고 (표현식이 될 수 있습니다), methodname 는 객체의 형에 의해 정의된 메서드의 이름입니다. 다른 형은 다른 메서드들을 정의합니다. 서로 다른 형들의 메서드는 모호함 없이 같은 이름을 가질 수 있습니다. (클래스 를 사용해서 여러분 자신의 형과 메서드를 정의하는 것이 가능합니다, 클래스 를 보세요) 예에 나오는 메서드 append() 는 리스트 객체들에 정의되어 있습니다; 요소를 리스트의 끝에 덧붙입니다. 이 예에서는 result = result + [a] 와 동등하지만, 더 효율적입니다.

4.7 함수 정의 더 보기

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

4.7.1 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:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok 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 끝내길 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 5 를 인쇄한다.

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
```

4.7.2 키워드 인자

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

```
def parrot(voltage, state='a stiff', action='voom', 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 아니다), 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 keyword argument 'a'
```

`**name` 형식의 마지막 형식 파라미터가 존재하면, 형식 파라미터들에 대응하지 않는 모든 키워드 인자들을 담은 딕셔너리 (typesmapping 를 보세요) 를 받습니다. 이것은 `*name` (다음 서브섹션에서 설명한다) 형식의 형식 파라미터와 조합될 수 있는데, 형식 파라미터 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.

4.7.3 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))
```

보통, 이 *가변 길이* 인자들은 형식 파라미터 목록의 마지막에 옵니다, 함수로 전달된 남은 입력 인자들 전부를 그러모으기 때문입니다. *args 파라미터 뒤에 등장하는 형식 파라미터들은 모두 〈키워드-전용〉 인자들인데, 위치 인자 대신 키워드 인자로만 사용될 수 있다는 뜻입니다.

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

4.7.4 인자 목록 언 패킹

인자들이 이미 리스트나 튜플에 있지만, 분리된 위치 인자들을 요구하는 함수 호출을 위해 언 패킹 해야 하는 경우 반대 상황이 벌어집니다. 예를 들어, 내장 range() 함수는 별도의 *start* 와 *stop* 인자를 기대합니다. 그것들이 따로 있지 않으면, 리스트와 튜플로부터 인자를 언 패킹하기 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 딕셔너리도 **-연산자를 써서 키워드 인자를 전달할 수 있습니다:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     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 !
```

4.7.5 람다 표현식

lambda 키워드들 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')]
```

4.7.6 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, 도큐멘테이션을 처리하는 도구들은 필요하면 들여쓰기를 제거합니다. 이것은 다음과 같은 관례를 사용합니다. 문자열의 첫줄 뒤에 오는 첫 번째 비어있지 않은 줄이 전체 도튜멘테이션 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.
```

4.7.7 함수 어노테이션

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

어노테이션은 함수의 `__annotations__` 어트리뷰트에 딕셔너리로 저장되고 함수의 다른 부분에는 아무런 영향을 미치지 않습니다. 파라미터 어노테이션은 파라미터 이름 뒤에 오는 콜론으로 정의되는데, 값을 구할 때 어노테이션의 값을 주는 표현식이 뒤따릅니다. 반환 값 어노테이션은 리터럴 `->` 와 그 뒤를 따르는 표현식으로 정의되는데, 파라미터 목록과 `def` 문의 끝을 나타내는 콜론 사이에 놓입니다. 다음 예에서 위치 인자, 키워드 인자, 반환 값이 어노테이트 됩니다:

```
>>> 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'
```

4.8 막간극: 코딩 스타일

이제 여러분은 파이썬의 더 길고, 더 복잡한 조각들을 작성하려고 합니다, 코딩 스타일 에 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:

- 들려 쓰기에 4-스페이스를 사용하고, 탭을 사용하지 마세요.

  4개의 스페이스는 작은 들여쓰기 (더 많은 중첩 도를 허락한다) 와 큰 들여쓰기 (읽기 쉽다) 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 `CamelCase`, 함수와 메서드의 경우 `lower_case_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.

CHAPTER 5

자료 구조

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*`)`
리스트의 끝에 항목을 더합니다. `a[len(a):] = [x]` 와 동등합니다.

`list.`**`extend`**`(`*iterable*`)`
리스트의 끝에 이터러블의 모든 항목을 덧붙여서 확장합니다. `a[len(a):] = iterable` 와 동등합니다.

`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* 인 첫 번째 항목을 삭제합니다. 그런 항목이 없으면 에러입니다.

`list.`**`pop`**`(`[*i*]`)`
리스트에서 주어진 위치에 있는 항목을 삭제하고, 그 항목을 돌려줍니다. 인덱스를 지정하지 않으면, `a.pop()` 은 리스트의 마지막 항목을 삭제하고 돌려줍니다. (메서드 시그니처에서 *i* 를 둘러싼 꺾쇠괄호는 파라미터가 선택적임을 나타냅니다. 그 위치에 꺾쇠괄호를 입력해야 한다는 뜻이 아닙니다. 이 표기법은 파이썬 라이브러리 레퍼런스에서 지주 등장합니다.)

`list.`**`clear`**`()`
리스트의 모든 항목을 삭제합니다. `del a[:]` 와 동등합니다.

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`
리스트에 있는 항목 중 값이 *x* 인 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`**`(`*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`**`()`
리스트의 얕은 사본을 돌려줍니다. `a[:]` 와 동등합니다.

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 a 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'
```

아마도 여러분은 `insert`, `remove`, `sort` 같은 메서드들이 리스트를 수정할 뿐 반환 값이 출력되지 않는 것을 알아챘을 것입니다 – 기본 `None` 을 돌려주고 있습니다.[1] 이것은 파이썬에서 모든 가변 자료 구조들에 적용되는 설계 원리입니다.

5.1.1 리스트를 스택으로 사용하기

리스트 메서드들은 리스트를 스택으로 사용하기 쉽게 만드는데, 마지막에 넣은 요소가 처음으로 꺼내지는 요소입니다 (《last-in, first-out》). 스택의 꼭대기에 항목을 넣으려면 `append()` 를 사용하세요. 스택의 꼭대기에서 값을 꺼내려면 명시적인 인덱스 없이 `pop()` 을 사용하세요. 예를 들어:

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

(다음 페이지에 계속)

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

(이전 페이지에서 계속)

```
7
>>> stack
[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 꺾쇠괄호로 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, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # 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]]
```

앞절에서 보았듯이, 중첩된 리스트 컴프리헨션은 뒤따르는 `for` 의 문맥에서 값이 구해집니다. 그래서 이 예는 다음과 동등합니다:

```
>>> 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 언 패킹 을 보세요.

5.2 del 문

리스트에서 값 대신에 인덱스를 사용해서 항목을 삭제하는 방법이 있습니다: del 문입니다. 이것은 값을 돌려주는 pop() 메서드와 다릅니다. del 문은 리스트에서 슬라이스를 삭제하거나 전체 리스트를 비우는 데도 사용될 수 있습니다(앞에서 빈 리스트를 슬라이스에 대입해서 했던 일입니다). 예를 들어:

```
>>> 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 자료형의 두 가지 예입니다(typesseq 를 보세요). 파이썬은 진화하는 언어이기 때문에, 다른 시퀀스 자료형이 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 보인다 하더라도, 이것들은 다른 상황에서 다른 목적으로 사용됩니다. 튜플은 불변 이고, 보통 이질적인 요소들의 시퀀스를 포함합니다. 요소들은 언 패킹 (이 섹션의 뒤에 나온다) 이나 인덱싱 (또는 네임드 튜플 의 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 튜플로 패킹 됩니다. 반대 연산 또한 가능합니다:

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

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

5.4 집합

파이썬은 집합 을 위한 자료 형도 포함합니다. 집합은 중복되는 요소가 없는 순서 없는 컬렉션입니다. 기본적인 용도는 멤버십 검사와 중복 엔트리 제거입니다. 집합 객체는 합집합, 교집합, 차집합, 대칭 차집합과 같은 수학적인 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'}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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 컴프리헨션 과 유사하게, 집합 컴프리헨션도 지원됩니다:

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

5.5 딕셔너리

파이썬에 내장된 또 하나의 쓸모있는 자료 형은 딕셔너리 입니다 (typesmapping 를 보세요). 딕셔너리는 종종 다른 언어들에서 《연관 메모리 (associative memories)》 나 《연관 배열 (associative arrays)》 의 형태로 발견됩니다. 숫자들로 인덱싱되는 시퀀스와 달리, 딕셔너리는 키 로 인덱싱되는데, 모든 불변형을 사용할 수 있습니다; 문자열과 숫자들은 항상 키가 될 수 있습니다. 튜플이 문자열, 숫자, 튜플들만 포함하면, 키로 사용될 수 있습니다; 튜플이 직접적이나 간접적으로 가변 객체를 포함하면, 키로 사용될 수 없습니다. 리스트는 키로 사용할 수 없는데, 리스트는 인덱스 대입, 슬라이스 대입, `append()` 나 `extend()` 같은 메서드들로 값이 수정될 수 있기 때문입니다.

딕셔너리를 (한 딕셔너리 안에서) 키가 중복되지 않는다는 제약 조건을 가진 키: 값 쌍의 순서 없는 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.

딕셔러리에 `list(d.keys())` 를 수행하면 딕셔너리에서 사용되고 있는 모든 키의 리스트를 돌려줍니다. 그 순서는 정해져 있지 않습니다 (정렬을 원하면 대신 `sorted(d.keys())` 를 사용하면 됩니다).[2] 하나의 키가 딕셔너리에 있는지 검사하려면, `in` 키워드들 사용하세요.

여기에 딕셔너리를 사용하는 조그마한 예가 있습니다:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
```

(다음 페이지에 계속)

[2] `d.keys()` 를 호출하면 딕셔너리 뷰 *(dictionary view)* 객체를 돌려줍니다. 이것은 멤버십 검사와 이터레이션 같은 연산들을 지원하지만, 그 내용은 원래 딕셔너리와 독립적이지 않습니다 – 이것은 뷰 일뿐입니다.

(이전 페이지에서 계속)

```
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, 'jack': 4098, 'guido': 4127}
```

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, 'jack': 4098, 'guido': 4127}
```

5.6 루프 테크닉

딕셔너리로 루핑할 때, items() 메서드를 사용하면 키와 거기에 대응하는 값을 동시에 얻을 수 있습니다.

```
>>> 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 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.

비교 연산자 in 과 not in 은 값이 시퀀스에 있는지 (없는지) 검사합니다. 연산자 is 와 is not 은 두 객체가 진짜로 같은 객체인지 비교합니다; 이것은 리스트와 같은 가변 객체에서만 의미가 있습니다. 모든 비교 연산자들은 같은 우선순위를 갖는데, 모든 산술 연산자들보다 낮습니다.

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, C와는 달리, 대입은 표현식 안에 등장할 수 없습니다. C 프로그래머들이 이 문제로 투덜거리지만, C 프로그램에서 흔히 마주치는 부류의 문제들을 회피하도록 합니다: == 를 사용할 표현식에 = 를 입력하는 실수.

5.8 시퀀스와 다른 형들을 비교하기

시퀀스 객체들은 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)
(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.

CHAPTER 6

모듈

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 스크립트 를 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 `fibo.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 b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

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

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

```
>>> import fibo
```

이렇게 한다고 fibo 에 정의된 함수들의 이름이 현재 심볼 테이블에 직접 들어가지는 않습니다; 오직 모듈 이름 fibo 만 들어갈 뿐입니다. 이 모듈 이름을 사용해서 함수들을 액세스할 수 있습니다:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[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)
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 초기화하는 데 사용됩니다. 이것들은 임포트 문에서 모듈 이름이 처음 등장할 때만 실행됩니다.[1] (이것들은 파일이 스크립트로 실행될 때도 실행됩니다.)

각 모듈은 자신만의 심볼 테이블을 갖고 있는데, 그 모듈에서 정의된 함수들의 전역 심볼 테이블로 사용됩니다. 그래서, 모듈의 저자는 사용자의 전역 변수와 우연히 충돌할 것을 걱정하지 않고 전역 변수를 사용할 수 있습니다. 반면에, 여러분이 무얼 하는지 안다면, 모듈의 함수를 참조하는데 사용된 것과 같은 표기법으로 모듈의 전역 변수들을 건드릴 수 있습니다, modname.itemname.

모듈은 다른 모듈들을 임포트할 수 있습니다. 모든 import 문들을 모듈의 처음에 놓는 것이 관례지만 반드시 그래야 하는 것은 아닙니다 (그 점에 관한 한 스크립트도 마찬가집니다). 임포트되는 모듈 이름은 임포트하는 모듈의 전역 심볼 테이블에 들어갑니다.

모듈에 들어있는 이름들을 직접 임포트하는 모듈의 심볼 테이블로 임포트하는 import 문의 변종이 있습니다. 예를 들어:

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

이것은 지역 심볼 테이블에 임포트되는 모듈의 이름을 만들지 않습니다 (그래서 이 예에서는, fibo 가 정의되지 않습니다).

모듈이 정의하는 모든 이름을 임포트하는 변종도 있습니다:

```
>>> from fibo import *
>>> fib(500)
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.

[1] 사실 함수 정의도 〈실행〉 되는 〈문장〉 입니다; 모듈 수준의 함수 정의를 실행하면 함수의 이름이 전역 심볼 테이블에 들어갑니다.

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.

```
>>> 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.

from을 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
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 스위트를 실행하기).

6.1.2 모듈 검색 경로

spam 이라는 이름의 모듈이 임포트될 때, 인터프리터는 먼저 그 이름의 내장 모듈을 찾습니다. 발견되지 않으면, 변수 sys.path 로 주어지는 디렉터리들에서 spam.py 라는 이름의 파일을 찾습니다. sys.path 는 이 위치들로 초기화됩니다:

- 입력 스크립트를 포함하는 디렉터리 (또는 파일이 지정되지 않았을 때는 현재 디렉터리).
- PYTHONPATH (디렉터리 이름들의 목록, 셸 변수 PATH 와 같은 문법).
- 설치 의존적인 기본값

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.
- There is more detail on this process, including a flow chart of the decisions, in **PEP 3147**.

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 플랫폼 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
'... '
>>> 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__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', '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',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', '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', 'fib', 'fibo', '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

패키지는 《점으로 구분된 모듈 이름》 를 써서 파이썬의 모듈 이름 공간을 구조화하는 방법입니다. 예를 들어, 모듈 이름 A.B 는 A 라는 이름의 패키지에 있는 B 라는 이름의 서브 모듈을 가리킵니다. 모듈의 사용이 다른 모듈의 저자들이 서로의 전역 변수 이름들을 걱정할 필요 없게 만드는 것과 마찬가지로, 점으로 구분된 모듈의 이름들은 NumPy 나 Pillow 과 같은 다중 모듈 패키지들의 저자들이 서로의 모듈 이름들을 걱정할 필요 없게 만듭니다.

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):

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

파이썬이 디렉터리를 패키지로 취급하게 만들기 위해서 `__init__.py` 파일이 필요합니다; 이렇게 하는 이유는 `string` 처럼 흔히 쓰는 이름의 디렉터리가, 의도하지 않게 모듈 검색 경로의 뒤에 등장하는 올바른 모듈들을 가리는 일을 방지하기 위함입니다. 가장 간단한 경우, `__init__.py` 는 그냥 빈 파일일 수 있지만, 패키지의 초기화 코드를 실행하거나 뒤에서 설명하는 `__all__` 변수를 설정할 수 있습니다.

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

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

이것은 서브 모듈 `sound.effects.echo` 를 로드합니다. 전체 이름으로 참조되어야 합니다.

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

서브 모듈을 임포트하는 다른 방법은 이렇습니다:

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

이것도 서브 모듈 `echo` 를 로드하고, 패키지 접두어 없이 사용할 수 있게 합니다. 그래서 이런 식으로 사용할 수 있습니다:

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

또 다른 방법은 원하는 함수나 변수를 직접 임포트하는 것입니다:

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

또다시, 이것은 서브 모듈 `echo` 를 로드하지만, 함수 `echofilter()` 를 직접 사용할 수 있게 만듭니다:

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

from package import item 를 사용할 때, item은 패키지의 서브 모듈 (또는 서브 패키지) 일 수도 있고 함수, 클래스, 변수 등 패키지에 정의된 다른 이름들일 수도 있음에 유의하세요. import 문은 먼저 item이 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 관례가 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"]
```

이것은 from sound.effects import * 이 sound.effects 패키지의 세 서브 모듈들을 임포트하게 됨을 의미합니다.

__all__ 이 정의되지 않으면, 문장 from sound.effects import * 은 패키지 sound.effects 의 모든 서브 모듈들을 현재 이름 공간으로 임포트 하지 않습니다; 이것은 오직 패키지 sound.effects 가 임포트 되도록 만들고 (__init__.py 에 있는 초기화 코드들이 수행될 수 있습니다), 그 패키지가 정의하는 이름들을 임포트 합니다. 이 이름들은 __init__.py 가 정의하는 모든 이름 (그리고 명시적으로 로드된 서브 모듈들) 을 포함합니다. 이 이름들에는 사전에 import 문으로 명시적으로 로드된 패키지의 서브 모듈들 역시 포함됩니다. 이 코드를 생각해봅시다:

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

이 예에서, echo 와 surround 모듈이 현재 이름 공간으로 임포트 되는데, from...import 문이 실행될 때 sound.effects 패키지에 정의되기 때문입니다. (__all__ 이 정의될 때도 마찬가집니다.)

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

패키지가 서브 패키지들로 구조화될 때 (예에서 나온 sound 패키지처럼), 이웃 패키지의 서브 모듈을 가리키는데 절대 임포트를 사용할 수 있습니다. 예를 들어, 모듈 sound.filters.vocoder 이 sound.effects 패키지의 echo 모듈이 필요하면, from sound.effects import echo 를 사용할 수 있습니다.

상대 임포트를 쓸 수도 있는데, from module import name 형태의 임포트 문을 사용합니다. 이 임포트는 상대 임포트에 수반되는 현재와 부모 패키지를 가리키기 위해 앞에 붙는 점을 사용합니다. 예를 들어, surround 모듈에서, 이렇게 사용할 수 있습니다:

```
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

패키지는 특별한 어트리뷰트 하나를 더 지원합니다, __path__. 이것은 패키지의 __init__.py 파일을 실행하기 전에, 이 파일이 들어있는 디렉터리의 이름을 포함하는 리스트로 초기화됩니다. 이 변수는 수정할 수 있습니다; 그렇게 하면 그 이후로 패키지에 포함된 모듈과 서브 패키지를 검색하는 데 영향을 주게 됩니다.

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

CHAPTER 7

입력과 출력

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

지금까지 우리는 값을 쓰는 두 가지 방법을 만났습니다: 표현식 문장 과 `print()` 함수입니다. (세 번째 방법은 파일 객체의 `write()` 메서드를 사용하는 것입니다; 표준 출력 파일은 `sys.stdout` 로 참조할 수 있습니다. 이것에 대한 자세한 정보는 라이브러리 레퍼런스를 보세요.)

종종 단순히 스페이스로 분리된 값들을 인쇄하기보다, 출력의 포맷을 좀 더 제어하고 싶기 마련입니다. 출력을 포매팅하는 두 가지 방법이 있습니다; 첫 번째 방법은 여러분 스스로 모든 문자열 처리를 하는 것입니다; 문자열 슬라이싱과 이어붙이기를 사용하면 여러분이 상상할 수 있는 어떤 배치라도 만들어 낼 수 있습니다. 문자열형은 문자열을 주어진 칼럼 폭으로 채워주는 편리한 연산들을 수행하는 메서드들을 제공합니다; 이것은 뒤에서 간단히 설명합니다. 두 번째 방법은 포맷 문자열 리터럴 이나 `str.format()` 메서드를 사용하는 것입니다.

`string` 모듈은 `Template` 클래스를 포함하는데, 값을 문자열에 치환하는 또 다른 방법을 제공합니다.

물론, 한가지 질문이 남아있습니다; 값을 어떻게 문자열로 변환하는가? 다행히도, 파이썬은 어떤 종류의 값이라도 문자열로 변환하는 방법을 갖고 있습니다; 그 값을 `repr()` 나 `str()` 함수로 전달하세요.

`str()` 함수는 어느 정도 사람이 읽기에 적합한 형태로 값의 표현을 돌려주게 되어있습니다. 반면에 `repr()` 은 인터프리터에 의해 읽힐 수 있는 형태를 만들게 되어있습니다 (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, 숫자들이나 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.'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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'))"
```

여기 제곱수와 세제곱수의 표를 쓰는 두 가지 방법이 있습니다:

```
>>> 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

>>> 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
```

(첫 번째 예에서, `print()` 의 동작 방식으로 인해 각 칼럼 사이에 스페이스 하나가 추가되었음에 유의하세요: 기본적으로 인자들 사이에 스페이스를 추가합니다.)

이 예는 문자열 객체의 `str.rjust()` 메서드를 시연하는데, 왼쪽에 스페이스를 채워서 주어진 폭으로 문자열을 우측 줄 맞춤합니다. 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 칼럼 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'
```

`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.
```

`'!a'` (`ascii()` 를 적용한다), `'!s'` (`str()` 을 적용합니다), `'!r'` (`repr()` 을 적용한다) 은 포맷 전에 값을 변환하는 데 사용됩니다:

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

선택적인 `':'` 과 포맷 지정자가 필드 이름 뒤에 올 수 있습니다. 이것으로 값이 포맷되는 방식을 더 정교하게 제어할 수 있습니다. 다음 예는 원주율을 소수점 이하 세 자리로 반올림합니다.

```
>>> import math
>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))
The value of PI is approximately 3.142.
```

`':'` 뒤에 정수를 전달하면 해당 필드의 최소 문자 폭이 됩니다. 표를 예쁘게 만들 때 편리합니다.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
Jack        ==>        4098
Dcab        ==>        7678
Sjoerd      ==>        4127
```

나누고 싶지 않은 정말 긴 포맷 문자열이 있을 때, 포맷할 변수들을 위치 대신에 이름으로 지정할 수 있다면 좋을 것입니다. 간단히 딕셔너리를 넘기고 키를 액세스하는데 꺾쇠괄호 `'[]'` 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
```

〈**〉 표기법을 사용해서 table을 키워드 인자로 전달해도 같은 결과를 얻을 수 있습니다.

```
>>> 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
```

이 방법은 모든 지역 변수들을 담은 딕셔너리를 돌려주는 내장 함수 `vars()` 와 함께 사용할 때 특히 쓸모가 있습니다.

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

7.1.1 예전의 문자열 포매팅

`%` 연산자도 문자열 포매팅에 사용될 수 있습니다. 왼쪽 인자를 오른쪽 인자에 적용되는 `sprintf()`-스타일 포맷 문자열로 해석하고, 이 포매팅 연산의 결과로 얻어지는 문자열을 돌려줍니다. 예를 들어:

```
>>> 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()` 은 파일 객체 를 돌려주고, 두 개의 인자를 주는 방식이 가장 많이 사용됩니다: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
```

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.

보통, 파일은 텍스트 모드 *(text mode)* 로 열리는데, 이 뜻은, 파일에 문자열을 읽고 쓰고, 파일에는 특정한 인코딩으로 저장된다는 것입니다. 인코딩이 지정되지 않으면 기본값은 플랫폼 의존적입니다 (`open()` 을 보세요). mode 에 덧붙여진 `'b'` 는 파일을 바이너리 모드 *(binary mode)* 로 엽니다: 이제 데이터는 바이트열 객체의 형태로 읽고 쓰입니다. 텍스트를 포함하지 않는 모든 파일에는 이 모드를 사용해야 합니다.

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. 혜택은 도중 예외가 발생하더라도 스위트가 종료될 때 파일이 올바르게 닫힌다는 것입니다. with 를 사용하는 것은 동등한 try-finally 블록을 쓰는 것에 비교해 훨씬 짧기도 합니다:

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

with 키워드를 사용하지 않으면, f.close() 를 호출해서 파일을 닫고 사용된 시스템 자원을 즉시 반납해야 합니다. 명시적으로 파일을 닫지 않으면, 파이썬의 가비지 수거기가 결국에는 객체를 파괴하고 여러분을 대신해서 파일을 닫게 되지만, 파일이 한동안 열린 상태로 남아있게 됩니다. 또 다른 위험은 다른 파이썬 구현들은 이 뒷정리를 서로 다른 시점에 수행한다는 것입니다.

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 모드 에서) 이나 바이트열 (바이너리 모드에서) 로 돌려줍니다. *size* 는 선택적인 숫자 인자다. *size* 가 생략되거나 음수면 파일의 내용 전체를 읽어서 돌려줍니다; 파일의 크기가 기계의 메모리보다 두 배 크다면 여러분이 감당할 문제입니다. 그렇지 않으면 최대 *size* 바이트를 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, from_what)` 를 사용합니다. 위치는 기준점에 *offset* 을 더해서 계산됩니다; 기준점은 *from_what* 인자로 선택합니다. *from_what* 값이 0이면 파일의 처음부터 측정하고, 1이면 현재 파일 위치를 사용하고, 2 는 파일의 끝을 기준점으로 사용합니다. *from_what* 은 생략될 수 있고, 기본값은 0이라서 파일의 처음을 기준점으로 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 0뿐입니다. 그 밖의 다른 *offset* 값은 정의되지 않은 결과를 낳습니다.

파일 객체는 `isatty()` 나 `truncate()` 같은 몇 가지 메서드를 더 갖고 있는데, 덜 자주 사용됩니다; 파일 객체에 대한 완전한 안내는 라이브러리 레퍼런스를 참조하세요.

7.2.2 json 으로 구조적인 데이터를 저장하기

문자열은 파일에 쉽게 읽고 쓸 수 있습니다. 숫자는 약간의 수고를 해야 하는데, read() 메서드가 문자열만을 돌려주기 때문입니다. 이 문자열을 int() 같은 함수로 전달해야만 하는데, '123' 같은 문자열을 받고 숫자 값 123을 돌려줍니다. 중첩된 리스트나 딕셔너리 같은 더 복잡한 데이터를 저장하려고 할 때, 수작업으로 파싱하고 직렬화하는 것이 까다로울 수 있습니다.

사용자가 반복적으로 복잡한 데이터형을 파일에 저장하는 코드를 작성하고 디버깅하도록 하는 대신, 파이썬은 JSON (JavaScript Object Notation) 이라는 널리 쓰이는 데이터 교환 형식을 사용할 수 있게 합니다. json 이라는 표준 모듈은 파이썬 데이터 계층을 받아서 문자열 표현으로 바꿔줍니다; 이 절차를 직렬화 *(serializing)* 라고 부릅니다. 문자열 표현으로부터 데이터를 재구성하는 것은 역 직렬화 *(deserializing)* 라고 부릅니다. 직렬화와 역 직렬화 사이에서, 객체를 표현하는 문자열은 파일이나 데이터에 저장되거나 네트워크 연결을 통해 원격 기계로 전송될 수 있습니다.

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

dump() 라는 dumps() 함수의 변종은 객체를 텍스트 파일 로 직렬화합니다. 그래서 f 가 쓰기를 위해 열린 텍스트 파일 이면, 이렇게 할 수 있습니다:

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

객체를 다시 디코드하려면, f 가 읽기를 위해 열린 텍스트 파일 객체일 때:

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

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.

CHAPTER 8

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
```

파서는 문제가 되는 줄을 다시 보여주고 줄에서 에러가 감지된 가장 앞의 위치를 가리키는 작은 '화살표'를 표시합니다. 에러는 화살표 앞에 오는 토큰이 원인입니다(또는 적어도 그곳에서 감지되었습니다): 이 예에서, 에러는 함수 `print()` 에서 감지되었는데, 그 앞에 콜론(`':'`) 이 빠져있기 때문입니다. 파일 이름과 줄 번호가 인쇄되어서, 입력이 스크립트로부터 올 때 찾을 수 있도록 합니다.

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>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

에러 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.

에러 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 그 들의 의미를 나열하고 있습니다.

8.3 예외 처리하기

선택한 예외를 처리하는 프로그램을 만드는 것이 가능합니다. 다음 예를 보면, 올바를 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.
- try 절을 실행하는 동안 예외가 발생하면, 절의 남은 부분들을 건너뜁니다. 그런 다음 형이 `except` 키워드 뒤에 오는 예외 이름과 매치되면, 그 except 절이 실행되고, 그런 다음 실행은 `try` 문 뒤로 이어집니다.
- except 절에 있는 예외 이름들과 매치되지 않는 예외가 발생하면, 외부에 있는 `try` 문으로 전달됩니다; 처리기가 발견되지 않으면, 처리되지 않은 예외 이고 위에서 보인 것과 같은 메시지를 출력하면서 실행이 멈춥니다.

각기 다른 예외에 대한 처리기를 지정하기 위해, `try` 문은 하나 이상의 except 절을 가질 수 있습니다. 최대 하나의 처리기가 실행됩니다. 처리기는 해당하는 try 절에서 발생한 예외만 처리할 뿐 같은 `try` 문의 다른 처리기가 일으킨 예외를 처리하지는 않습니다. except 절은 괄호가 있는 튜플로 여러 개의 예외를 지정할 수 있습니다, 예를 들어:

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

except 절에 있는 클래스는 예외와 같은 클래스이거나 베이스 클래스일 때 매치됩니다 (하지만 다른 방식으로는 매치되지 않습니다 — 자식 클래스를 나열한 except 절은 베이스 클래스와 매치되지 않습니다). 예를 들어, 다음과 같은 코드는 B, C, D를 그 순서대로 인쇄합니다:

```
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")
```

except 절이 뒤집히면 (except B 가 처음에 오도록), B, B, B를 인쇄하게 됨에 주의하세요 — 처음으로 매치되는 절이 실행됩니다.

마지막 except 절은 예외 이름을 생략할 수 있는데, 와일드카드 역할을 합니다. 이것을 사용할 때는 극도의 주의를 필요로 합니다. 이런 식으로 실제 프로그래밍 에러를 가리기 쉽기 때문입니다! 에러 메시지를 인쇄한 후에 예외를 다시 일으키는데 사용될 수도 있습니다 (호출자도 예외를 처리할 수 있도록):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try … except 문은 선택적인 *else* 절 을 갖는데, 있다면 모든 except 절 뒤에와야 합니다. try 절이 예외를 일으키지 않을 때 실행되어야만 하는 코드에 유용합니다. 예를 들어:

```
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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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.

예외가 발생할 때, 연관된 값을 가질 수 있는데, 예외의 인자 라고도 알려져 있습니다. 인자의 존재와 형은 예외 형에 의존적입니다.

except 절은 예외 이름 뒤에 변수를 지정할 수 있습니다. 변수는 인자들이 instance.args 에 저장된 예외 인스턴스에 연결됩니다. 편의를 위해, 예외 인스턴스는 __str__() 를 정의해서, .args 를 참조하지 않고도 인자들을 직접 인쇄할 수 있습니다. 예외를 일으키기 전에 인스턴스를 먼저 만들고 필요한 어트리뷰트들을 추가할 수도 있습니다.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     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
```

예외가 인자를 가지면, 처리되지 않은 예외 메시지의 마지막 부분(〈상세 명세〉)에 인쇄됩니다.

예외 처리기는 단지 try 절에 직접 등장하는 예외뿐만 아니라, try 절에서 (간접적으로라도) 호출되는 내부 함수들에서 발생하는 예외들도 처리합니다. 예를 들어:

```
>>> 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>
NameError: HiThere
```

raise 에 제공하는 단일 인자는 발생시킬 예외를 가리킵니다. 예외 인스턴스이거나 예외 클래스 (Exception 를 계승하는 클래스) 이어야 합니다. 예외 클래스가 전달되면, 묵시적으로 인자 없이 생성자를 호출해서 인스턴스를 만듭니다:

```
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>
NameError: HiThere
```

8.5 사용자 정의 예외

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

예외 클래스는 다른 클래스들이 할 수 있는 어떤 것도 가능하도록 정의될 수 있지만, 보통은 간단하게 유지합니다. 종종 예외 처리기가 에러에 관한 정보를 추출할 수 있도록 하기 위한 몇 가지 어트리뷰트들을 제공하기만 합니다. 여러 가지 서로 다른 에러들을 일으킬 수 있는 모듈을 만들 때, 흔히 사용되는 방식은 모듈에서 정의되는 예외들의 베이스 클래스를 정의한 후, 각기 다른 에러 조건마다 특정한 예외 클래스를 서브 클래스로 만드는 것입니다:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end in《Error》, similar to the naming of the standard exceptions.

많은 표준 모듈들은 그들이 정의하는 함수들에서 발생할 수 있는 그 자신만의 예외들을 정의합니다. 클래스에 관한 더 자세한 정보는 클래스 장에서 다룹니다.

8.6 뒷정리 동작 정의하기

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

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

finally 절 은 예외의 발생 여부와 관계없이 `try` 문을 떠날 때 항상 실행됩니다. `try` 절에서 예외가 발생하고 `except` 절에서 처리되지 않으면 (또는 `except` 나 `else` 절에서 발생하면), `finally` 절이 실행된 후에 다시 일으킵니다. `finally` 절은 `try` 문의 다른 모든 절에서 `break`, `continue`, `return` 문에 의해《빠져나가는 길에》도 실행됩니다. 더 복잡한 예는 이렇습니다:

```
>>> 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>
  File "<stdin>", line 3, in divide
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).

8.7 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 도큐멘테이션에서 이 사실을 설명합니다.

CHAPTER 9

클래스

클래스는 데이터와 기능을 함께 묶는 방법을 제공합니다. 새 클래스를 만드는 것은 객체의 새 형 을 만들어서, 그 형의 새 인스턴스 를 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 맴버 함수들은 *virtual* 입니다. 모듈라-3처럼, 객체의 매소드에서 그 객체의 멤버를 참조하는 줄임 표현은 없습니다: 메서드 함수는 그 객체를 표현하는 명시적인 첫 번째 인자를 선언하는데, 함수 호출 때 묵시적으로 제공됩니다. 스몰토크처럼, 클래스 자신도 객체입니다. 이것이 임포팅과 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 스몰토크나 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, 에일리어싱는 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 공간 은 이름에서 객체로 가는 매핑입니다. 대부분의 이름 공간은 현재 파이썬 딕셔너리로 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 어트리뷰트 라는 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![1]

어트리뷰트는 읽기 전용 이거나 쓰기 가능할 수 있습니다. 후자의 경우, 어트리뷰트에 대한 대입이 가능합니다. 모듈 어트리뷰트는 쓰기 가능합니다: `modname.the_answer = 42` 라고 쓸 수 있습니다. 쓰기 가능한 어트리뷰트는 `del` 문으로 삭제할 수도 있습니다. 예를 들어, `del modname.the_answer` 는 `modname` 라는 이름의 객체에서 어트리뷰트 `the_answer` 를 제거합니다.

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 그 들 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.

[1] 한 가지만 제외하고. 모듈 객체는 `__dict__` 라고 불리는 비밀스러운 읽기 전용 어트리뷰트를 갖는데, 모듈의 이름 공간을 구현하는데 사용하는 딕셔너리를 돌려줍니다; 이름 `__dict__` 는 어트리뷰트 이지만 전역 이름은 아닙니다. 명백하게, 이것을 사용하는 것은 이름 공간 구현의 추상화를 파괴하는 것이고, 사후 디버거와 같은 것들로만 제한되어야 합니다.

스코프는 이름 공간을 직접 액세스할 수 있는 파이썬 프로그램의 텍스트 적인 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, 적어도 세 개의 중첩된 스코프가 있습니다:

- 가장 먼저 검색되는, 가장 내부의 스코프는 지역 이름들을 포함합니다
- 둘러싸고 있는 함수들의 스코프는, 가장 가까이서 둘러싸는 스코프로부터 검색이 시작됩니다, 비 지역 (non-local) 이지만 비 전역 (non-global) 이름들을 포함합니다
- 마지막 직전의 스코프는 현재 모듈의 전역 이름들을 포함합니다
- (가장 나중에 검색되는) 가장 외부의 스코프는 내장 이름들을 포함하고 있는 이름 공간입니다.

이름을 global 로 선언하면, 모든 참조와 대입은 모듈의 전역 이름들을 포함하는 중간 스코프로 바로 갑니다. 가장 내부의 스코프 바깥에서 발견되는 변수들을 재연결하려면, `nonlocal` 키워드를 사용할 수 있습니다; nonlocal 로 선언되지 않으면, 그 변수들은 읽기 전용입니다 (그런 변수에 쓰려고 하면 단순히 가장 내부의 스코프에 새 지역 변수를 만들게 되어, 같은 이름의 바깥 변수를 바꾸지 않고 남겨둡니다).

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` 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"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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 변수들로의 대입은 이 새 이름 공간으로 갑니다. 특히, 함수 정의는 새 함수의 이름을 이곳에 연결합니다.

클래스 정의가 (끝을 통해) 정상적으로 끝날 때, 클래스 객체 가 만들어집니다. 이것은 기본적으로 클래스 정의 때문에 만들어진 이름 공간의 내용물들을 감싸는 싸개입니다; 다음 섹션에서 클래스 객체에 대해 더 배우게 됩니다. 원래의 지역 스코프가 (클래스 정의에 들어가기 직전에 유효하던 것) 다시 사용되고, 클래스 객체는 클래스 정의 헤더에서 주어진 클래스 이름 (예에서 `ClassName`) 으로 여기에 연결됩니다.

9.3.2 클래스 객체

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

어트리뷰트 참조 는 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'
```

`MyClass.i` 와 `MyClass.f` 는 올바른 어트리뷰트 참조고, 각기 정수와 함수 객체를 돌려줍니다. 클래스 어트리뷰트는 대입할 수도 있어서, 대입을 통해 `MyClass.i` 의 값을 변경할 수 있습니다. `__doc__` 도 역시 올바른 어트리뷰트고, 클래스에 속하는 독스트링을 돌려줍니다: `"A simple example class"`.

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

```
x = MyClass()
```

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

인스턴스 만들기 연산 (클래스 객체 《호출하기》) 은 빈 객체를 만듭니다. 많은 클래스는 특정한 초기 상태로 커스터마이즈된 인스턴스로 객체를 만드는 것을 좋아합니다. 그래서 클래스는 이런 식으로 `__init__()` 라는 이름의 특수 메서드 정의할 수 있습니다:

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

클래스가 `__init__()` 메서드를 정의할 때, 클래스 인스턴스 만들기는 새로 만들어진 클래스 인스턴스에 대해 자동으로 `__init__()` 를 호출합니다. 그래서 이 예에서, 새 초기화된 인스턴스를 이렇게 얻을 수 있습니다:

```
x = MyClass()
```

물론, `__init__()` 메서드는 더 높은 유연성을 위해 인자들을 가질 수 있습니다. 그 경우, 클래스 인스턴스 만들기 연산자로 주어진 인자들은 `__init__()` 로 전달됩니다. 예를 들어,

```
>>> 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

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

데이터 어트리뷰트 는 스몰토크의《인스턴스 변수》에, C++ 의《데이터 멤버》에 해당합니다. 데이터 어트리뷰트는 선언될 필요 없습니다; 지역 변수처럼, 처음 대입될 때 태어납니다. 예를 들어, `x` 가 위에서 만들어진 `MyClass` 의 인스턴스면, 다음과 같은 코드 조각은 트레이스 없이 값 `16` 을 인쇄합니다:

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

다른 인스턴스 어트리뷰트 참조는 메서드 입니다. 메서드는 객체에《속하는》함수입니다. (파이썬에서, 메서드 라는 용어는 클래스 인스턴스에만 사용되지 않습니다; 다른 객체 형들도 메서드를 가질 수 있습니다. 예를 들어, 리스트 객체는 append, insert, remove, sort 등과 같은 메서드들을 갖습니다. 하지만, 앞으로의 논의에서, 명시적으로 언급하지 않는 한, 메서드 라는 용어를 클래스 인스턴스 객체의 메서드에만 사용할 것입니다.)

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()
```

`MyClass` 예에서, 이것은 문자열 `'hello world'` 를 돌려줍니다. 하지만, 메서드를 즉시 호출할 필요는 없습니다: `x.f` 는 메서드 객체고, 저장된 후에 호출될 수 있습니다. 예를 들어:

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

는 영원히 계속 `hello world` 를 인쇄합니다.

메서드가 호출될 때 정확히 어떤 일이 일어날까? `f()` 의 함수 정의가 인자를 지정했음에도 불구하고, 위에서 `x.f()` 는 인자 없이 호출된 것을 알아챘을 것입니다. 인자는 어떻게 된 걸까? 확실히 파이썬은 인자를 필요로 하는 함수를 인자 없이 호출하면 예외를 일으킵니다 – 인자가 실제로는 사용되지 않는다 해도…

실제로, 여러분은 답을 짐작할 수 있습니다: 메소드의 특별함은 인스턴스 객체가 함수의 첫 번째 인자로 전달된다는 것입니다. 우리 예에서, 호출 `x.f()` `` 은 **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.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. 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, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the 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:

    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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 기타 주의사항들

데이터 어트리뷰트는 같은 이름의 메서드 어트리뷰트를 덮어씁니다; 의도하지 않은 이름 충돌(큰 프로그램에서 찾기 어려운 버그를 만듭니다)을 피하려면, 충돌의 기회를 최소화하는 어떤 종류의 규칙을 사용하는 것이 현명합니다. 가능한 규칙에는 메서드 이름을 대문자로 시작하는 것, 데이터 어트리뷰트의 이름에 작고 특별한 문자열 (아마도 밑줄 하나)을 앞에 붙이는 것, 메서드에는 동사를 데이터 어트리뷰트에는 명사를 쓰는 것들이 있습니다.

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 불변성 들을 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
```

이제 `f`, `g`, `h` 는 모두 함수 객체를 가리키는 클래스 `C` 의 어트리뷰트고, 결과적으로 이것들은 모두 `C` 의 인스턴스들의 메서드입니다 — `h` 는 정확히 `g` 와 동등합니다. 이런 방식은 프로그램의 독자들에게 혼란을 주기만 한다는 점에 주의하세요.

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 붙일만한 가치가 없을 것입니다. 파생 클래스 정의의 문법은 이렇게 생겼습니다:

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

이름 BaseClassName 은 파생 클래스 정의를 포함하는 스코프에 정의되어 있어야 합니다. 베이스 클래스 이름의 자리에 다른 임의의 표현식도 허락됩니다. 예를 들어, 베이스 클래스가 다른 모듈에 정의되어 있을 때 유용합니다:

```
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>
```

대부분의 목적상, 가장 간단한 경우에, 부모 클래스로부터 상속된 어트리뷰트들의 검색을 깊이 우선으로, 왼쪽에서 오른쪽으로, 계층 구조에서 겹치는 같은 클래스를 두 번 검색하지 않는 것으로 생각할 수 있습니다. 그래서, 어트리뷰트가 `DerivedClassName` 에서 발견되지 않으면, `Base1` 에서 찾고, 그다음 (재귀적으로) `Base1` 의 베이스 클래스들을 검색합니다. 거기에서도 발견되지 않으면, `Base2` 에서 찾고, 이런 식으로 계속합니다.

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.

동적인 순서가 필요한 이유는, 모든 다중 상속의 경우는 하나나 그 이상의 다이아몬드 관계 (적어도 부모 클래스 중 하나가 가장 바닥 클래스들로부터 여러 경로를 통해 액세스 되는 경우) 를 만들기 때문입니다. 예를 들어, 모든 클래스는 `object` 를 계승하기 때문에, 모든 다중 상속은 `object` 에 이르는 여러 경로를 제공합니다. 베이스 클래스들이 여러 번 액세스 되지 않게 하려고, 동적인 알고리즘이 검색 순서를 선형화하는데, 각 클래스에서 지정된 왼쪽에서 오른쪽으로 가는 순서를 보존하고, 각 부모를 오직 한 번만 호출하고, 단조적 (부모들의 우선순위에 영향을 주지 않으면서 서브 클래스를 만들 수 있다는 의미입니다) 이도록 만듭니다. 모두 함께 사용될 때, 이 성질들은 다중 상속으로 신뢰성 있고 확장성 있는 클래스들을 설계할 수 있도록 만듭니다. 더 자세한 내용은, https://www.python.org/download/releases/2.3/mro/ 를 보세요.

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.

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)
```

The above example would work even if MappingSubclass were to introduce a __update identifier since it is replaced with _Mapping__update in the Mapping class and _MappingSubclass__update in the MappingSubclass class respectively.

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

때로 몇몇 이름 붙은 데이터 항목들을 함께 묶어주는 파스칼의 《record》 나 C의 《struct》 와 유사한 데이터형을 갖는 것이 쓸모 있습니다. 빈 클래스 정의가 훌륭히 할 수 있는 일입니다:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

특정한 추상적인 데이터형을 기대하는 파이썬 코드 조각은, 종종 그 데이터형의 메서드를 흉내 내는 클래스를 대신 전달받을 수 있습니다. 예를 들어, 파일 객체로부터 데이터를 포맷하는 함수가 있을 때, 대신 문자열 버퍼에서 데이터를 읽는 메서드 read() 와 readline() 을 제공하는 클래스를 정의한 후 인자로 전달할 수 있습니다.

인스턴스 메서드 객체도 어트리뷰트를 갖습니다: m.__self__ 는 메서드 m() 과 결합한 인스턴스 객체이고, m.__func__ 는 메서드에 상응하는 함수 객체입니다.

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
<iterator object at 0x00A1DB50>
>>> 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
```

이터레이터 프로토콜의 뒤에 있는 메커니즘을 살펴보면, 여러분의 클래스에 이터레이터 동작을 쉽게 추가할 수 있습니다. __next__() 메서드를 가진 객체를 돌려주는 __iter__() 메서드를 정의합니다. 클래스가 __next__() 를 정의하면, __iter__() 는 그냥 self 를 돌려줄 수 있습니다.

```
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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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

제너레이터 는 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
```

제너레이터로 할 수 있는 모든 것은 앞 절에서 설명했듯이 클래스 기반 이터레이터로도 할 수 있습니다. 제너레이터가 간단한 이유는 `__iter__()` 와 `__next__()` 메서드가 저절로 만들어지기 때문입니다.

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 비교해 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, 꺾쇠괄호 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

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> 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']
```

CHAPTER 10

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:\\Python36'
>>> 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

일반적인 유틸리티 스크립트는 종종 명령행 인자를 처리해야 할 필요가 있습니다. 이 인자들은 sys 모듈의 *argv* 어트리뷰트에 리스트로 저장됩니다. 예를 들어, 명령행에서 `python demo.py one two three` 를 실행하면 다음과 같은 결과가 출력됩니다:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

getopt 모듈은 유닉스 `getopt()` 함수의 규칙을 사용해서 *sys.argv* 를 처리합니다. 더 강력하고 유연한 명령행 처리는 argparse 모듈에서 제공됩니다.

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'\bf[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

math 모듈은 부동 소수점 연산을 위한 하부 C 라이브러리 함수들에 대한 액세스를 제공합니다.

```
>>> 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
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 URL에서 데이터를 읽어오는 urllib.request 와 메일을 보내는 smtplib 입니다:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # Decoding the binary data to text.
...         if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
... 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
```

`timeit` 의 정밀도와는 대조적으로, `profile` 과 `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 도큐멘테이션을 개선하고, doctest 모듈이 도큐멘테이션에서 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:

- `xmlrpc.client` 와 `xmlrpc.server` 모듈은 원격 프로시저 호출을 구현하는 일을 거의 사소한 일로 만듭니다. 모듈의 이름에도 불구하고, XML에 대한 직접적인 지식이나 처리가 필요하지 않습니다.
- The `email` package is a library for managing email messages, including MIME and other **RFC 2822**-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- `json` 패키지는 널리 사용되는 데이터 교환 형식을 파싱하기위한 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 액세스 될 수 있는 퍼시스턴트 데이터베이스를 제공합니다.
- 국제화는 `gettext`, `locale`, 그리고 `codecs` 패키지를 포함한 많은 모듈에 의해 지원됩니다.

CHAPTER 11

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 축약 된 디스플레이를 위해 커스터마이즈된 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
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 포매팅하는 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("%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 커스터마이즈할 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 이스케이프된 $ 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 누락 된 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 = '%'
>>> 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

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 록, 이벤트, 조건 변수 및 세마포를 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 그램, 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 로드 할 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:/python36/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.

array 모듈은 array() 객체를 제공합니다. 이 객체는 등질적인 데이터만을 저장하고 보다 조밀하게 저장하는 리스트와 같습니다. 다음 예제는 파이썬 int 객체의 일반 리스트의 경우처럼 항목당 16바이트를 사용하는 대신에, 2바이트의 부호 없는 이진 숫자 (형 코드 "H") 로 저장된 숫자 배열을 보여줍니다:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
26932
>>> a[1:3]
array('H', [10, 700])
```

collections 모듈은 deque() 객체를 제공합니다. 이 객체는 왼쪽에서 더 빠르게 추가/팝하지만 중간에서의 조회는 더 느려진 리스트와 같습니다. 이 객체는 대기열 및 넓이 우선 트리 검색을 구현하는 데 적합합니다:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> 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 10진 부동 소수점 산술

decimal 모듈은 10진 부동 소수점 산술을 위한 Decimal 데이터형을 제공합니다. 내장 float 이진 부동 소수점 구현과 비교할 때, 클래스는 특히 다음과 같은 것들에 유용합니다

- 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. Decimal은 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
>>> sum([0.1]*10) == 1.0
False
```

decimal 모듈은 필요한 만큼의 정밀도로 산술을 제공합니다:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

CHAPTER 12

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 설치가 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

가상 환경을 만들고 관리하는 데 사용되는 모듈은 `venv` 라고 합니다. `venv` 는 보통 여러분이 사용할 수 있는 최신 버전의 파이썬을 설치합니다. 시스템에 여러 버전의 파이썬이 있는 경우, `python3` 또는 원하는 버전을 실행하여 특정 파이썬 버전을 선택할 수 있습니다.

가상 환경을 만들려면, 원하는 디렉터리를 결정하고, `venv` 모듈을 스크립트로 실행하는데 디렉터리 경로를 명령행 인자로 전달합니다:

```
python3 -m venv tutorial-env
```

존재하지 않는다면 `tutorial-env` 디렉터리를 만들고, 그 안에 파이썬 인터프리터의 사본, 표준 라이브러리 및 다양한 지원 파일이 들어있는 디렉터리들을 만듭니다.

가상 환경을 만들었으면, 가상 환경을 활성화할 수 있습니다.

윈도우에서 이렇게 실행합니다:

```
tutorial-env\Scripts\activate.bat
```

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']
>>>
```

12.3 pip로 패키지 관리하기

pip 라는 프로그램을 사용하여 패키지를 설치, 업그레이드 및 제거할 수 있습니다. 기본적으로 `pip` 는 파이썬 패키지 색인 (Python Package Index), <https://pypi.org>, 에서 패키지를 설치합니다. 웹 브라우저에서 파이썬 패키지 색인을 살펴보거나, `pip` 의 제한된 검색 기능을 사용할 수 있습니다:

```
(tutorial-env) $ pip search astronomy
skyfield               - Elegant astronomy for Python
gary                   - Galactic astronomy and gravitational dynamics.
novas                  - The United States Naval Observatory NOVAS astronomy library
astroobs               - Provides astronomy ephemeris to plan telescope observations
PyAstronomy            - A collection of astronomy related tools for Python.
...
```

`pip` 는 《search》, 《install》, 《uninstall》, 《freeze》 등 많은 부속 명령을 갖고 있습니다. (`pip` 에 대한 완전한 문서는 installing-index 지침을 보면 됩니다.)

패키지 이름을 지정하여 최신 버전의 패키지를 설치할 수 있습니다:

```
(tutorial-env) $ 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) $ 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
```

이 명령을 다시 실행하면, pip 는 요청한 버전이 이미 설치되어 있음을 알리고, 아무것도 하지 않습니다. 다른 버전 번호를 지정해서 그 버전을 얻거나 pip install --upgrade 를 실행하여 패키지를 최신 버전으로 업그레이드할 수 있습니다:

```
(tutorial-env) $ pip install --upgrade requests
Collecting requests
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
```

pip uninstall 다음에 하나 이상의 패키지 이름이 오면 가상 환경에서 패키지가 제거됩니다.

pip show 는 특정 패키지에 대한 정보를 표시합니다:

```
(tutorial-env) $ 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:
```

pip list 는 가상 환경에 설치된 모든 패키지를 표시합니다:

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

pip freeze 는 설치된 패키지의 비슷한 목록을 만들지만, pip install 이 기대하는 형식을 사용합니다. 일반적인 규칙은 이 목록을 requirements.txt 파일에 넣는 것입니다:

```
(tutorial-env) $ 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) $ 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 에는 더 많은 옵션이 있습니다. pip 에 대한 완전한 문서는 installing-index 지침을 참고하세요. 패키지를 작성했을 때 파이썬 패키지 색인에서 사용할 수 있게 하려면, distributing-index 지침을 참고하세요.

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, CGI 프로그램을 작성하고, 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. 이 웹 사이트는 유럽, 일본 및 호주와 같이 전 세계 여러 곳에 미러가 만들어집니다. 지리적 위치에 따라 미러가 기본 사이트보다 빠를 수도 있습니다.

- https://docs.python.org: 파이썬의 도큐멘테이션에 빠르게 액세스할 수 있습니다.

- https://pypi.org: 이전에 치즈 가게 (Cheese Shop) 로도 불렸던 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.

- http://www.pyvideo.org 는 콘퍼런스 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.

파이썬 관련 질문 및 문제 보고의 경우, 뉴스 그룹 *comp.lang.python* 에 게시하거나 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.

CHAPTER 14

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 도큐멘테이션이 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 비교해 엄청난 발전입니다; 그러나, 몇 가지 희망 사항이 남아 있습니다: 이어지는 줄에서 적절한 들여쓰기가 제안된다면 좋을 것입니다 (파서는 다음에 들여쓰기 토큰이 필요한지 알고 있습니다). 완료 메커니즘은 인터프리터의 심볼 테이블을 사용할 수 있습니다. 매치되는 괄호, 따옴표 등을 검사 (또는 제안) 하는 명령도 유용할 것입니다.

꽤 오랫동안 사용됐던 개선된 대화형 인터프리터는 IPython 인데, 탭 완성, 객체 탐색 및 고급 히스토리 관리 기능을 갖추고 있습니다. 또한, 철저하게 커스터마이즈해서 다른 응용 프로그램에 내장할 수 있습니다. 비슷한 또 다른 개선된 대화형 환경은 bpython 입니다.

CHAPTER 15

부동 소수점 산술: 문제점 및 한계

부동 소수점 숫자는 컴퓨터 하드웨어에서 밑(base)이 2인(이진) 소수로 표현됩니다. 예를 들어, 소수

```
0.125
```

는 1/10 + 2/100 + 5/1000의 값을 가지며, 같은 방식으로 이진 소수

```
0.001
```

는 값 0/2 + 0/4 + 1/8을 가집니다. 이 두 소수는 같은 값을 가지며, 유일한 차이점은 첫 번째가 밑이 10인 분수 표기법으로 작성되었고 두 번째는 밑이 2라는 것입니다.

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 근사 될 뿐입니다.

이 문제는 먼저 밑 10에서 따져보는 것이 이해하기 쉽습니다. 분수 1/3을 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 1/3이 될 수 없지만, 점점 더 1/3에 가까운 근사치가 됩니다.

같은 방식으로, 아무리 많은 자릿수의 숫자를 사용해도, 십진수 0.1은 이진 소수로 정확하게 표현될 수 없습니다. 이진법에서, 1/10은 무한히 반복되는 소수입니다

```
0.0001100110011001100110011001100110011001100110011...
```

유한 한 비트 수에서 멈추면, 근삿값을 얻게 됩니다. 오늘날 대부분 기계에서, float는 이진 분수로 근사 되는 데, 최상위 비트로부터 시작하는 53비트를 분자로 사용하고, 2의 거듭제곱 수를 분모로 사용합니다. 1/10의 경우, 이진 분수는 `3602879701896397 / 2 ** 55` 인데, 실제 값 1/10과 거의 같지만 정확히 같지는 않습니다.

많은 사용자는 값이 표시되는 방식 때문에 근사를 인식하지 못합니다. 파이썬은 기계에 저장된 이진 근삿값의 진짜 십진 값에 대한 십진 근삿값을 인쇄할 뿐입니다. 대부분 기계에서, 만약 파이썬이 0.1로 저장된 이진 근삿값의 진짜 십진 값을 출력한다면 다음과 같이 표시해야 합니다

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

이것은 대부분 사람이 유용하다고 생각하는 것보다 많은 숫자이므로, 파이썬은 반올림된 값을 대신 표시하여 숫자를 다룰만하게 만듭니다

```
>>> 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 `0.1000000000000001` 및 `0.1000000000000000055511151231257827021181583404541015625` 는 모두 `3602879701896397 / 2 ** 55` 로 근사 됩니다. 이 십진 값들이 모두 같은 근삿값을 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 3.1부터, 이제 파이썬(대부분 시스템에서)이 가장 짧은 것을 선택할 수 있으며, 단순히 `0.1` 만 표시합니다.

이것이 이진 부동 소수점의 본질임에 주목하세요: 파이썬의 버그는 아니며, 여러분의 코드에 있는 버그도 아닙니다. 하드웨어의 부동 소수점 산술을 지원하는 모든 언어에서 같은 종류의 것을 볼 수 있습니다(일부 언어는 기본적으로 혹은 모든 출력 모드에서 차이를 표시하지 않을 수 있지만).

좀 더 만족스러운 결과를 얻으려면, 문자열 포매팅을 사용하여 제한된 수의 유효 숫자를 생성할 수 있습니다:

```
>>> 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 기곗값의 표시 를 반올림하고 있습니다.

하나의 환상은 다른 환상을 낳을 수 있습니다. 예를 들어, 0.1은 정확히 1/10이 아니므로, 0.1의 세 개를 합한 것 역시 정확히 0.3이 아닙니다:

```
>>> .1 + .1 + .1 == .3
False
```

또한, 0.1은 1/10의 정확한 값에 더 가까워질 수 없고, 0.3도 3/10의 정확한 값에 더 가까워질 수 없으므로, `round()` 함수로 미리 반올림하는 것은 도움이 되지 않습니다:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

숫자를 의도한 정확한 값에 더 가깝게 만들 수는 없지만, round() 함수는 사후 반올림에 유용하여 부정확한 값을 가진 결과를 서로 비교할 수 있게 합니다:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

이진 부동 소수점 산술은 이처럼 많은 놀라움을 안겨줍니다. 《0.1》의 문제는 아래의 《표현 오류》 섹션에서 자세하게 설명합니다. 부동 소수점의 위험 은 다른 흔히 만나는 놀라움에 대해 더욱 완전한 설명을 제공합니다.

끝이 가까이 오면 말하듯이, 《쉬운 답은 없습니다.》 아직, 부동 소수점수를 지나치게 경계할 필요는 없습니다! 파이썬 float 연산의 에러는 부동 소수점 하드웨어에서 상속된 것이고, 대부분 기계에서는 연산당 2**53분의 1을 넘지 않는 규모입니다. 이것은 대부분 작업에서 필요한 수준 이상입니다. 하지만, 십진 산술이 아니며 모든 float 연산에 새로운 반올림 에러가 발생할 수 있다는 점을 명심해야 합니다.

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 사용해보세요.

정확한 산술의 또 다른 형태는 유리수를 기반으로 산술을 구현하는 fractions 모듈에 의해 지원됩니다 (따라서 1/3과 같은 숫자는 정확하게 나타낼 수 있습니다).

부동 소수점 연산을 많이 하는 사용자면 Numerical Python 패키지와 SciPy 프로젝트에서 제공하는 수학 및 통계 연산을 위한 다른 많은 패키지를 살펴봐야 합니다. <https://scipy.org> 를 보세요.

파이썬은 여러분이 float의 정확한 값을 진짜로 알아야 하는 드문 경우를 지원할 수 있는 도구들을 제공합니다. float.as_integer_ratio() 메서드는 float의 값을 분수로 표현합니다:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

비율은 정확한 값이기 때문에, 원래 값을 손실 없이 다시 만드는 데 사용할 수 있습니다:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

float.hex() 메서드는 float를 16진수(밑이 16이다)로 표현하는데, 컴퓨터에 저장된 정확한 값을 줍니다:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

이 정확한 16진수 표현은 float 값을 정확하게 재구성하는 데 사용할 수 있습니다:

```
>>> 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.

또 다른 유용한 도구는 math.fsum() 함수입니다. 이 함수는 합산 동안 정밀도 상실을 완화합니다. 누적 합계에 값이 더해지면서 《잃어버린 숫자들》을 추적합니다. 최종 합계에 영향을 주는 지점까지 에러가 누적되지 않아서 전체적인 정확도에 차이를 만들 수 있습니다:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

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.

왜 그럴까? 1/10은 이진 소수로 정확히 표현할 수 없습니다. 오늘날 (2000년 11월) 거의 모든 기계는 IEEE-754 부동 소수점 산술을 사용하고, 거의 모든 플랫폼은 파이썬 float를 IEEE-754 《배정밀도》에 매핑합니다. 754 배정밀도는 53비트의 정밀도가 포함되어 있어서, 입력 시 컴퓨터는 0.1을 *J*/2**N* 형태의 가장 가까운 분수로 변환하려고 노력합니다. 여기서 *J* 는 정확히 53비트를 포함하는 정수입니다.:

```
1 / 10 ~= J / (2**N)
```

를

```
J ~= 2**N / 10
```

로 다시 쓰고, *J*가 정확히 *53* 비트 *(">= 2**52" 이지만 "< 2**53" 다)* 임을 고려하면, *N* 의 최적값은 56입니다:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

즉, 56은 *J* 가 정확히 53비트가 되도록 만드는 *N* 의 유일한 값입니다. *J* 의 가능한 값 중 가장 좋은 것은 반올림한 몫입니다:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

나머지가 10의 절반보다 크므로, 가장 가까운 근삿값은 올림 해서 얻어집니다:

```
>>> q+1
7205759403792794
```

따라서 754 배정밀도로 1/10 에 가장 가까운 근삿값은 다음과 같습니다:

```
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 1/10일 수는 없습니다!

따라서 컴퓨터는 결코 1/10을 《보지》 못합니다: 볼 수 있는 것은 위에서 주어진 정확한 분수, 얻을 수 있는 최선의 754 배정밀도 근삿값입니다:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

그 분수에 10**55를 곱하면, 55개의 십진 숫자를 볼 수 있습니다.:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

이는 컴퓨터에 저장된 정확한 숫자가 십진수 0.1000000000000000055511151231257827021181583404541015625와 같음을 의미합니다. 전체 십진법 값을 표시하는 대신, 많은 언어(이전 버전의 파이썬 포함)는 결과를 17개의 유효숫자로 반올림합니다:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

fractions 와 decimal 모듈은 이 계산을 쉽게 만듭니다:

```
>>> 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('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

CHAPTER 16

부록

16.1 대화형 모드

16.1.1 에러 처리

에러가 발생하면 인터프리터는 에러 메시지와 스택 트레이스를 인쇄합니다. 대화형 모드에서는 기본 프롬프트로 돌아갑니다; 파일로부터 입력이 왔을 때는, 스택 트레이스를 인쇄한 후 0이 아닌 종료 상태로 종료합니다. (`try` 문에서 `except` 절에 의해 처리되는 예외는 이 문맥에서 에러가 아닙니다.) 일부 에러는 무조건 치명적이며 0이 아닌 종료 상태의 종료를 유발합니다; 이것은 내부 불일치와 메모리 부족으로 인한 경우에 적용됩니다. 모든 에러 메시지는 표준 에러 스트림에 기록됩니다. 실행된 명령의 정상 출력은 표준 출력에 기록됩니다.

기본 또는 보조 프롬프트에 인터럽트 문자 (일반적으로 `Control-C` 또는 `Delete`)를 입력하면 입력을 취소하고 기본 프롬프트로 돌아갑니다.[1] 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 python3.5
```

(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
```

[1] GNU Readline 패키지에 있는 문제가 이것을 방해할 수 있습니다.

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

파이썬은 커스터마이즈할 수 있는 두 가지 훅을 제공합니다: sitecustomize 와 usercustomize. 어떻게 작동하는지 보려면, 먼저 여러분의 사용자 site-packages 디렉터리의 위치를 찾아야 합니다. 파이썬을 시작하고 다음 코드를 실행합니다:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.5/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 는 같은 방식으로 작동하지만, 일반적으로 전역 site-packages 디렉터리에 컴퓨터 관리자가 만들고, usercustomize 전에 임포트됩니다. 자세한 내용은 site 모듈의 도큐멘테이션을 보세요.

APPENDIX A

용어집

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

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

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

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

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

annotation A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

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

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

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

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

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

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

용어집의 파라미터 항목과 FAQ 질문 인자와 파라미터의 차이 와 **PEP 362** 도 보세요.

asynchronous context manager (비동기 컨텍스트 관리자) `__aenter__()` 와 `__aexit__()` 메서드를 정의함으로써 `async with` 문에서 보이는 환경을 제어하는 객체. **PEP 492** 로 도입되었다.

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 다르다.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

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

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

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

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

asynchronous iterator (비동기 이터레이터) An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492**.

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

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

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

binary file (바이너리 파일) 바이트열류 객체들 을 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 있다.

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

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

어떤 연산들은 바이너리 데이터가 가변적일 필요가 있다. 이런 경우에 도큐멘테이션은 종종 《읽고-쓰기 바이트열류 객체》라고 표현한다. 가변 버퍼 객체의 예로는 bytearray 와 bytearray 의 memoryview 가 있다. 다른 연산들은 바이너리 데이터가 불변 객체 (《읽기 전용 바이트열류 객체》) 에 저장되도록 요구한다; 이런 것들의 예로는 bytes 와 bytes 객체의 memoryview 가 있다.

bytecode (바이트 코드) 파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현이다. 바이트 코드는 .pyc 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만든다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있다). 이 《중간 언어》는 각 바이트 코드에 대응하는 기계를 실행하는 가상 기계 에서 실행된다고 말한다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 한다.

바이트 코드 명령어들의 목록은 dis 모듈 도큐멘테이션에 나온다.

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

class variable A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

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

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

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

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

coroutine (코루틴) 코루틴은 서브루틴의 더 일반화된 형태다. 서브루틴은 한 지점에서 진입하고 다른 지점에서 탈출한다. 코루틴은 여러 다른 지점에서 진입하고, 탈출하고, 재개할 수 있다. 이것들은 async def 문으로 구현할 수 있다. **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 있다. 이것들은 **PEP 492** 에 의해 도입되었다.

CPython 파이썬 프로그래밍 언어의 규범적인 구현인데, python.org 에서 배포된다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어《CPython》이 사용된다.

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

데코레이터 문법은 단지 편의 문법일 뿐이다. 다음 두 함수 정의는 의미상으로 동등하다:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰인다. 데코레이터에 대한 더 자세한 내용은 함수 정의 와 클래스 정의 의 도큐멘테이션을 보면 된다.

descriptor (디스크립터) 메서드 `__get__()` 이나 `__set__()` 이나 `__delete__()` 를 정의하는 객체. 클래스 어트리뷰트가 디스크립터일 때, 어트리뷰트 조회는 특별한 연결 작용을 일으킨다. 보통, *a.b* 를 읽거나, 쓰거나, 삭제하는데 사용할 때, *a* 의 클래스 딕셔너리에서 *b* 라고 이름 붙여진 객체를 찾는다. 하지만 *b* 가 디스크립터면, 해당하는 디스크립터 메서드가 호출된다. 디스크립터를 이해하는 것은 파이썬에 대한 깊은 이해의 열쇠인데, 함수, 메서드, 프라퍼티, 클래스 메서드, 스태틱 메서드, 슈퍼클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문이다.

디스크립터의 메서드들에 대한 자세한 내용은 descriptors 에 나온다.

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

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

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

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

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

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

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

f-string (f-문자열) `'f'` 나 `'F'` 를 앞에 붙인 문자열 리터럴들을 흔히 《f-문자열》이라고 부르는데, 포맷 문자열 리터럴 의 줄임말이다. **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 저장장치나 통신 장치 (예를 들어, 표준 입출력, 인-메모리 버퍼, 소켓, 파이프, 등등) 에 대한 액세스를 중계할 수 있다. 파일 객체는 파일류 객체 *(file-like objects)* 나 스트림 *(streams)* 이라고도 불린다.

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

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

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

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

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

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

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

function annotation (함수 어노테이션) An *annotation* of a function parameter or return value.

Function annotations are usually used for *type hints*: for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

Function annotation syntax is explained in section function.

See *variable annotation* and **PEP 484**, which describe this functionality.

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

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

```
>>> 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 수행한다. 가비지 수거기는 `gc` 모듈을 사용해서 제어할 수 있다.

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

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

generator iterator (제너레이터 이터레이터) 제너레이터 함수가 만드는 객체.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression (제너레이터 표현식) 이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 `for` 표현식과 생략 가능한 `if` 표현식이 뒤에 붙는 일반 표현식 처럼 보인다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어낸다:

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

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

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

GIL 전역 인터프리터 록 을 보세요.

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

하지만, 어떤 확장 모듈들은, 표준이나 제삼자 모두, 압축이나 해싱 같은 계산 집약적인 작업을 수행할 때는 GIL 을 반납하도록 설계되었다. 또한, I/O를 할 때는 항상 GIL 을 반납한다.

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

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

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

모든 파이썬의 불변 내장 객체들은 해시 가능하다. (리스트나 딕셔너리 같은) 가변 컨테이너들은 그렇지 않다. 사용자 정의 클래스의 인스턴스 객체들은 기본적으로 해시 가능하다. (자기 자신을 제외하고는) 모두 다르다고 비교되고, 해시값은 `id()` 로 부터 만들어진다.

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

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

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

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

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

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

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

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

인터프리터 종료의 주된 원인은 실행되는 __main__ 모듈이나 스크립트가 실행을 끝내는 것이다.

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

이터러블은 for 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 (zip(), map(), …) 에 사용될 수 있다. 이터러블 객체가 내장 함수 iter() 에 인자로 전달되면, 그 객체의 이터레이터를 돌려준다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효하다. 이터러블을 사용할 때, 보통은 iter() 를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없다. for 문은 이것들을 여러분을 대신해서 자동으로 해주는데, 루프를 도는 동안 이터레이터를 잡아둘 이름 없는 변수를 만든다. 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 돌려준다. 더 이상의 데이터가 없을 때는 대신 StopIteration 예외를 일으킨다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 __next__() 메서드 호출은 StopIteration 예외를 다시 일으키기만 한다. 이터레이터는 이터레이터 객체 자신을 돌려주는 __iter__() 메서드를 가질 것이 요구되기 때문에, 이터레이터는 이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드다. (list 같은) 컨테이너 객체는 iter() 함수로 전달하거나 for 루프에 사용할 때마다 새 이터레이터를 만든다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만든다.

typeiter 에 더 자세한 내용이 있다.

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

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

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

keyword argument (키워드 인자) 인자 를 보세요.

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

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

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

list (리스트) A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is O(1).

list comprehension (리스트 컴프리헨션) 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 만든다. if 절은 생략할 수 있다. 생략하면, range(256) 에 있는 모든 요소가 처리된다.

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

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

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

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

metaclass (메타 클래스) 클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 딕셔너리, 베이스 클래스들의 목록을 만든다. 메타 클래스는 이 세 인자를 받아서 클래스를 만드는 책임을 진다. 대부분의 객체 지향형 프로그래밍 언어들은 기본 구현을 제공한다. 파이썬을 특별하게 만드는 것은 커스텀 메타 클래스를 만들 수 있다는 것이다. 대부분 사용자에게는 이 도구가 전혀 필요 없지만, 필요가 생길 때, 메타 클래스는 강력하고 우아한 해법을 제공한다. 어트리뷰트 액세스의 로깅(logging), 스레드 안전성의 추가, 객체 생성 추적, 싱글톤 구현과 많은 다른 작업에 사용됐다.

metaclasses 에서 더 자세한 내용을 찾을 수 있다.

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

method resolution order (메서드 결정 순서) 메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 The Python 2.3 Method Resolution Order 를 보면 된다.

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

패키지 도 보세요.

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

MRO 메서드 결정 순서 를 보세요.

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

named tuple (네임드 튜플) 인덱싱할 수 있는 요소들을 이름 붙은 어트리뷰트로도 액세스할 수 있는 모든 튜플류 클래스 (예를 들어, `time.localtime()` 은 *year* 가 `t[0]` 처럼 인덱스로도, `t.tm_year` 처럼 어트리뷰트로도 액세스할 수 있는 튜플류 객체를 돌려준다.)

네임드 튜플은 `time.struct_time` 같은 내장형일 수도, 일반 클래스 정의로 만들 수도 있다. 모든 기능이 구현된 네임드 튜플을 팩토리 함수 `collections.namedtuple()` 로도 만들 수 있다. 마지막 접근법은 `Employee(name='jones', title='programmer')` 와 같은 스스로 문서로 만드는 repr 과 같은 확장 기능도 자동 제공한다.

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

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

모듈 도 보세요.

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

new-style class (뉴스타일 클래스) 지금은 모든 클래스 객체에 사용되고 있는 클래스 버전의 예전 이름. 초기의 파이썬 버전에서는, 오직 뉴스타일 클래스만 `__slots__`, 디스크립터, 프라퍼티, `__getattribute__()`, 클래스 메서드, 스태틱 메서드와 같은 파이썬의 새롭고 다양한 기능들을 사용할 수 있었다.

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

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

정규 패키지 와 이름 공간 패키지 도 보세요.

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

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

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

- 위치-전용 *(positional-only)*: 위치로만 제공될 수 있는 인자를 지정한다. 파이썬은 위치-전용 파라미터를 정의하는 문법을 갖고 있지 않다. 하지만, 어떤 매장 함수들은 위치-전용 파라미터를 갖는다 (예를 들어, `abs()`).

- 키워드-전용 *(keyword-only)*: 키워드로만 제공될 수 있는 인자를 지정한다. 키워드-전용 파라미터는 함수 정의의 파라미터 목록에서 앞에 하나의 가변-위치 파라미터나 `*` 를 그대로 포함해서 정의할 수 있다. 예를 들어, 다음에서 *kw_only1* 와 *kw_only2*:

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

- 가변-위치 *(var-positional)*: (다른 파라미터들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정한다. 이런 파라미터는 파라미터 이름에 `*` 를 앞에 붙여서 정의될 수 있다, 예를 들어 다음에서 *args*:

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

- 가변-키워드 *(var-keyword)*: (다른 파라미터들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정한다. 이런 파라미터는 파라미터 이름에 `**` 를 앞에 붙여서 정의될 수 있다, 예를 들어 위의 예 에서 *kwargs*.

파라미터는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있다.

인자 용어집 항목, 인자와 파라미터의 차이 에 나오는 FAQ 질문, `inspect.Parameter` 클래스, function 섹션, **PEP 362** 도 보세요.

path entry (경로 엔트리) 경로 기반 파인더 가 임포트할 모듈들을 찾기 위해 참고하는 임포트 경로 상의 하나의 장소.

path entry finder (경로 엔트리 파인더) `sys.path_hooks` 에 있는 콜러블 (즉, 경로 엔트리 훅) 이 돌려주는 파인더 인데, 주어진 경로 엔트리 로 모듈을 찾는 방법을 알고 있다.

경로 엔트리 파인더들이 구현하는 메서드들은 `importlib.abc.PathEntryFinder` 에 나온다.

path entry hook (경로 엔트리 훅) `sys.path_hook` 리스트에 있는 콜러블인데, 특정 경로 엔트리 에서 모듈을 찾는 법을 알고 있다면 경로 엔트리 파인더 를 돌려준다.

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

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

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

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

PEP 1 참조하세요.

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

positional argument (위치 인자) 인자 를 보세요.

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

잠정 API에서조차도, 과거 호환성이 유지되지 않는 변경은《최후의 수단》으로 여겨진다 - 모든 식별된 문제들에 대해 과거 호환성을 유지하는 해법을 찾으려는 모든 시도가 선행된다.

이 절차는 표준 라이브러리가 오랜 시간 동안 잘못된 설계 오류에 발목 잡히지 않고 발전할 수 있도록 만든다. 더 자세한 내용은 **PEP 411** 를 보면 된다.

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

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

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

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

더 깔끔한, 파이썬다운 방법은 이렇다:

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

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

```
>>> 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 의미한다, 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 반납된다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소다. `sys` 모듈은 특정 객체의 참조 횟수를 돌려주는 `getrefcount()` 을 정의한다.

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

이름 공간 패키지 도 보세요.

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

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 있다. `dict` 또한 `__getitem__()` 과 `__len__()` 을 지원하지만, 조회에 정수 대신 임의의 불변 키를 사용하기 때문에 시퀀스가 아니라 매핑으로 취급된다는 것에 주의해야 한다.

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

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 만든다. `variable_name[1:3:5]` 처럼, `[]` 안에서 여러 개의 숫자를 콜론으로 분리한다. 꺾쇠괄호 (서브 스크립트) 표기법은 내부적으로 `slice` 객체를 사용한다.

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

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

struct sequence (구조체 시퀀스) A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

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

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

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

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

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

type alias A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying *type hints*. For example:

```
from typing import List, Tuple

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

could be made more readable like this:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

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

See `typing` and **PEP 484**, which describe this functionality.

type hint An *annotation* that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See `typing` and **PEP 484**, which describe this functionality.

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) An *annotation* of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

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

Variable annotations are usually used for *type hints*: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section annassign.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

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 실행한다.

Zen of Python (파이썬 젠) 파이썬 디자인 원리와 철학들의 목록인데, 인어를 이해하고 사용하는 데 도움이 된다. 이 목록은 대화형 프롬프트에서 《import this》를 입력하면 보인다.

APPENDIX B

About these documents

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the reporting-bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!

APPENDIX C

History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

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

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

참고: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.6.15

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

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.6.15 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2021 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.6.15 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.6.15 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.6.15.

4. PSF is making Python 3.6.15 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.6.15 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.6.15
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.6.15, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

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

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

8. By copying, installing or otherwise using Python 3.6.15, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

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

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

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

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


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

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
  ---------------------------------------------------------------------
 /                      Copyright (c) 1996.                            \
|          The Regents of the University of California.                 |
|                        All rights reserved.                           |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

C.3.5 Cookie management

The `http.cookies` module contains the following notice:

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

               All Rights Reserved

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

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

C.3.6 Execution tracing

The `trace` module contains the following notice:

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

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

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

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

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


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

C.3.7 UUencode and UUdecode functions

The `uu` module contains the following notice:

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

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.8 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

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

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

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

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

C.3.9 test_epoll

The `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.3.10 Select kqueue

The `select` module contains the following notice for the kqueue interface:

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

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

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

C.3.11 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski〉 implementation of Dan Bernstein's SipHash24 algorithm. The contains the following note:

```
<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>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

C.3.12 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.13 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

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

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

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

   /* ====================================================================
    * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    *
    * 1. Redistributions of source code must retain the above copyright
    *    notice, this list of conditions and the following disclaimer.
    *
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in
    *    the documentation and/or other materials provided with the
    *    distribution.
    *
    * 3. All advertising materials mentioning features or use of this
    *    software must display the following acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
    *
    * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
    *    endorse or promote products derived from this software without
    *    prior written permission. For written permission, please contact
    *    openssl-core@openssl.org.
    *
    * 5. Products derived from this software may not be called "OpenSSL"
    *    nor may "OpenSSL" appear in their names without prior written
    *    permission of the OpenSSL Project.
    *
    * 6. Redistributions of any form whatsoever must retain the following
    *    acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
    *
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

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

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   * All rights reserved.
   *
   * This package is an SSL implementation written
   * by Eric Young (eay@cryptsoft.com).
   * The implementation was written so as to conform with Netscapes SSL.
   *
   * This library is free for commercial and non-commercial use as long as
   * the following conditions are aheared to.  The following conditions
   * apply to all code found in this distribution, be it the RC4, RSA,
   * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
   * included with this distribution is covered by the same copyright terms
   * except that the holder is Tim Hudson (tjh@cryptsoft.com).
   *
   * Copyright remains Eric Young's, and as such any Copyright notices in
   * the code are not to be removed.
   * If this package is used in a product, Eric Young should be given attribution
   * as the author of the parts of the library used.
   * This can be in the form of a textual message at program startup or
   * in documentation (online or textual) provided with the package.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   * 1. Redistributions of source code must retain the copyright
   *    notice, this list of conditions and the following disclaimer.
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in the
   *    documentation and/or other materials provided with the distribution.
   * 3. All advertising materials mentioning features or use of this software
   *    must display the following acknowledgement:
   *    "This product includes cryptographic software written by
   *     Eric Young (eay@cryptsoft.com)"
   *    The word 'cryptographic' can be left out if the rouines from the library
   *    being used are not cryptographic related :-).
   * 4. If you include any Windows specific code (or a derivative thereof) from
   *    the apps directory (application code) you must include an acknowledgement:
   *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
   *
   * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
   * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
   * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
   * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
   * SUCH DAMAGE.
   *
   * The licence and distribution terms for any publically available version or
   * derivative of this code cannot be changed.  i.e. this code cannot simply be
   * copied and put under another distribution licence
   * [including the GNU Public Licence.]
   */
```

C.3.14 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured --with-system-expat:

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

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

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

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

C.3.15 libffi

The _ctypes extension is built using an included copy of the libffi sources unless the build is configured --with-system-libffi:

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

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

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

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

C.3.16 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

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

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

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

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

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

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

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

C.3.17 cfuhash

The implementation of the hash table used by the `tracemalloc` is based on the cfuhash project:

```
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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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.3.18 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

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

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

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

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

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

APPENDIX D

저작권

파이썬과 이 도큐멘테이션은:

전체 라이센스 및 사용 권한 정보는 *History and License* 에서 제공한다.

색인

Non-alphabetical

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P