Python Tutorial

출시 버전 *2.7.18*

Guido van Rossum
and the Python development team

5월 20, 2020

Contents

1 입맛 돋우기 **3**

2 파이썬 인터프리터 사용하기 **5**
2.1 인터프리터 실행하기 5
2.2 인터프리터와 환경 6

3 파이썬의 간략한 소개 **9**
3.1 파이썬을 계산기로 사용하기 10
3.2 프로그래밍으로의 첫걸음 18

4 기타 제어 흐름 도구 **19**
4.1 `if` 문 19
4.2 `for` 문 20
4.3 `range()` 함수 20
4.4 루프의 `break` 와 `continue` 문, 그리고 `else` 절 21
4.5 `pass` 문 22
4.6 함수 정의하기 22
4.7 함수 정의 더 보기 24
4.8 막간극: 코딩 스타일 28

5 자료 구조 **29**
5.1 리스트 더 보기 29
5.2 `del` 문 34
5.3 튜플과 시퀀스 35
5.4 집합 36
5.5 딕셔너리 37
5.6 루프 테크닉 38
5.7 조건 더 보기 39
5.8 시퀀스와 다른 형들을 비교하기 39

6 모듈 **41**
6.1 모듈 더 보기 42
6.2 표준 모듈들 45
6.3 `dir()` 함수 45
6.4 패키지 46

7 입력과 출력 **51**

7.1 장식적인 출력 포매팅 . . . 51
7.2 파일을 읽고 쓰기 . . . 54

8 에러와 예외 59
8.1 문법 에러 . . . 59
8.2 예외 . . . 59
8.3 예외 처리하기 . . . 60
8.4 예외 일으키기 . . . 62
8.5 사용자 정의 예외 . . . 63
8.6 뒷정리 동작 정의하기 . . . 64
8.7 미리 정의된 뒷정리 동작들 . . . 65

9 클래스 67
9.1 이름과 객체에 관한 한마디 . . . 67
9.2 파이썬 스코프와 이름 공간 . . . 68
9.3 클래스와의 첫 만남 . . . 69
9.4 기타 주의사항들 . . . 73
9.5 상속 . . . 74
9.6 Private Variables and Class-local References . . . 76
9.7 잡동사니 . . . 77
9.8 Exceptions Are Classes Too . . . 77
9.9 이터레이터 . . . 78
9.10 제너레이터 . . . 79
9.11 제너레이터 표현식 . . . 80

10 표준 라이브러리 둘러보기 81
10.1 운영 체제 인터페이스 . . . 81
10.2 파일 와일드카드 . . . 82
10.3 명령행 인자 . . . 82
10.4 에러 출력 리디렉션과 프로그램 종료 . . . 82
10.5 문자열 패턴 매칭 . . . 82
10.6 수학 . . . 83
10.7 인터넷 액세스 . . . 83
10.8 날짜와 시간 . . . 84
10.9 데이터 압축 . . . 84
10.10 성능 측정 . . . 84
10.11 품질 관리 . . . 85
10.12 배터리가 포함됩니다 . . . 85

11 표준 라이브러리 둘러보기 — 2부 87
11.1 출력 포매팅 . . . 87
11.2 템플릿 . . . 88
11.3 바이너리 데이터 레코드 배치 작업 . . . 89
11.4 다중 스레딩 . . . 90
11.5 로깅 . . . 90
11.6 약한 참조 . . . 91
11.7 리스트 작업 도구 . . . 91
11.8 10진 부동 소수점 산술 . . . 92

12 이제 뭘 하지? 95

13 대화형 입력 편집 및 히스토리 치환 97
13.1 Line Editing . . . 97
13.2 History Substitution . . . 97
13.3 Key Bindings . . . 98

13.4 대화형 인터프리터 대안 99

14 부동 소수점 산술: 문제점 및 한계 **101**
14.1 표현 오류 103

15 부록 **105**
15.1 대화형 모드 105

A 용어집 **107**

B About these documents **117**
B.1 Contributors to the Python Documentation 117

C History and License **119**
C.1 History of the software 119
C.2 Terms and conditions for accessing or otherwise using Python 120
C.3 Licenses and Acknowledgements for Incorporated Software 123

D 저작권 **135**

색인 **137**

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

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

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines, the Python installation is usually placed in `C:\Python27`, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python27
```

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.

The interpreter's line-editing features usually aren't very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing; see Appendix 대화형 입력 편집 및 히스토리 치환 for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

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.

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.

All command-line options are described in using-on-general.

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.

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
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 = 1
>>> 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

By default, Python source files are treated as encoded in ASCII. To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- 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 python
# -*- 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 - 5.0*6) / 4
5.0
>>> 8 / 5.0
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.

The return type of a division (/) operation depends on its operands. If both operands are of type int, *floor division* is performed and an int is returned. If either operand is a float, classic division is performed and a float is returned. The // operator is also provided for doing floor division no matter what the operands are. The remainder can be calculated with the % operator:

```
>>> 17 / 3  # int / int -> int
5
>>> 17 / 3.0  # int / float -> float
5.666666666666667
>>> 17 // 3.0  # explicit floor division discards the fractional part
5.0
>>> 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:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

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.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is

[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.

enclosed in single quotes. The `print` statement produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"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.

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

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

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:

typesseq Strings, and the Unicode strings described in the next section, are examples of *sequence types*, and support the common operations supported by such types.

string-methods Both strings and Unicode strings support a large number of methods for basic transformations and searching.

formatstrings `str.format()` 으로 문자열을 포맷하는 방법에 대한 정보.

string-formatting The old formatting operations invoked when strings and Unicode strings are the left operand of the `%` operator are described in more detail here.

3.1.3 Unicode Strings

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data (see http://www.unicode.org/) and integrates well with the existing string objects, providing auto-conversions where necessary.

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as `i18n` — `'i'` + 18 characters + `'n'`) of software. Unicode solves these problems by defining one code page for all scripts.

Creating Unicode strings in Python is just as simple as creating normal strings:

```
>>> u'Hello World !'
u'Hello World !'
```

The small `'u'` in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python *Unicode-Escape* encoding. The following example shows how:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

The escape sequence `\u0020` indicates to insert the Unicode character with the ordinal value 0x0020 (the space character) at the given position.

Other characters are interpreted by using their respective ordinal values directly as Unicode ordinals. If you have literal strings in the standard Latin-1 encoding that is used in many Western countries, you will find it convenient that the lower 256 characters of Unicode are the same as the 256 characters of Latin-1.

For experts, there is also a raw mode just like the one for normal strings. You have to prefix the opening quote with ⟨ur⟩ to have Python use the *Raw-Unicode-Escape* encoding. It will only apply the above `\uXXXX` conversion if there is an uneven number of backslashes in front of the small ⟨u⟩.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\u0020World !'
```

The raw mode is most useful when you have to enter lots of backslashes, as can be necessary in regular expressions.

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

The built-in function `unicode()` provides access to all registered Unicode codecs (COders and DECoders). Some of the more well known encodings which these codecs can convert are *Latin-1*, *ASCII*, *UTF-8*, and *UTF-16*. The latter two are variable-length encodings that store each Unicode character in one or more bytes. The default encoding is normally set to ASCII, which passes through characters in the range 0 to 127 and rejects any other characters with an error. When a Unicode string is printed, written to a file, or converted with `str()`, conversion takes place using this default encoding.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal␣
↪not in range(128)
```

To convert a Unicode string into an 8-bit string using a specific encoding, Unicode objects provide an `encode()` method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

If you have data in a specific encoding and want to produce a corresponding Unicode string from it, you can use the `unicode()` function with the encoding name as the second argument.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 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]
```

Lists also supports operations like concatenation:

```
>>> 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
['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 0과1이 대입됩니다. 마지막 줄에서 다시 사용되는데, 대입이 어느 하나라도 이루어지기 전에 우변의 표현식들이 모두 계산됩니다. 우변의 표현식은 왼쪽부터 오른쪽으로 가면서 순서대로 계산됩니다.
- `while` 루프는 조건(여기서는: `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.
- The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

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

  A trailing comma avoids the newline after the output:

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

  Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

CHAPTER 4

기타 제어 흐름 도구

Besides the `while` statement just introduced, Python uses the usual flow control statements known from other languages, with some twists.

4.1 `if` 문

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

```
>>> x = int(raw_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']
```

4.3 range() 함수

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; range(10) generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ⟨step⟩):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
1 had
2 a
3 little
4 lamb
```

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

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
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,
...         a, b = b, a+b
...
>>> # 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 참조

입니다).[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
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print`:

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

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

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

[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).

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, complaint='Yes or no, please!'):
    while True:
        ok = raw_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 IOError('refusenik user')
        print complaint
```

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,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

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

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

하지만 다음과 같은 호출들은 모두 올바르지 않습니다:

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

함수 호출에서, 키워드 인자는 위치 인자 뒤에 나와야 합니다. 전달된 모든 키워드 인자는 함수가 받아들이는 인자 중 하나와 맞아야 하며 (예를 들어, actor 는 parrot 함수의 올바른 인자가 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
    keys = sorted(keywords.keys())
    for kw in keys:
        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.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

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))
```

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:

```
>>> range(3, 6)             # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> 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,
...     print "if you put", voltage, "volts through it.",
...     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

There are emerging conventions about the content and formatting of documentation strings.

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.
```

4.8 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).

- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

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

`list.`**`extend`**`(`*L*`)`
Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`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*`)`
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`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.`**`index`**`(`*x*`)`
Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

`list.`**`count`**`(`*x*`)`
리스트에서 *x* 가 등장하는 횟수를 돌려줍니다.

`list.sort`(*cmp=None, key=None, reverse=False*)
리스트의 항목들을 제자리에서 정렬합니다 (인자들은 정렬 커스터마이제이션에 사용될 수 있습니다. 설명은 `sorted()` 를 보세요).

`list.reverse`()
Reverse the elements of the list, in place.

리스트 메서드 대부분을 사용하는 예:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

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

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()
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 Functional Programming Tools

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true. If *sequence* is a `str`, `unicode` or `tuple`, the result will be of the same type; otherwise, it is always a `list`. For example, to compute a sequence of numbers divisible by 3 or 5:

```
>>> def f(x): return x % 3 == 0 or x % 5 == 0
...
>>> filter(f, range(2, 25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or `None` if some sequence is shorter than another). For example:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Don't use this example's definition of `sum()`: since summing numbers is such a common need, a built-in function `sum(sequence)` is already provided, and works exactly like this.

5.1.4 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]
```

We can obtain the same result with:

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

This is also equivalent to `squares = map(lambda x: x**2, range(10))`, but it's more concise and readable.

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)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1,2,3]:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     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']
```

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:

```
>>> 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
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

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']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

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

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sorted()` function to it). To check whether a single key is in the dictionary, use the `in` keyword.

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}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

`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

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(xrange(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
```

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

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

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

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

```
(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)
```

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc.[1] Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

[1] The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.

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,
        a, b = b, a+b

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
```

This imports all names except those beginning with an underscore (_).

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.

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

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

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

`from`을 써서 비슷한 효과를 낼 때도 사용할 수 있습니다:

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

참고: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `reload()`, e.g. `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:

- the directory containing the input script (or the current directory).
- PYTHONPATH (디렉터리 이름들의 목록, 셸 변수 PATH 와 같은 문법).
- the installation-dependent default.

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

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called spam.pyc exists in the directory where spam.py is found, this is assumed to contain an already-《byte-compiled》 version of the module spam. The modification time of the version of spam.py used to create spam.pyc is recorded in spam.pyc, and the .pyc file is ignored if these don't match.

Normally, you don't need to do anything to create the spam.pyc file. Whenever spam.py is successfully compiled, an attempt is made to write the compiled version to spam.pyc. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting spam.pyc file will be recognized as invalid and thus ignored later. The contents of the spam.pyc file are platform independent, so a Python module directory can be shared by machines of different architectures.

전문가를 위한 몇 가지 팁

- When the Python interpreter is invoked with the -O flag, optimized code is generated and stored in .pyo files. The optimizer currently doesn't help much; it only removes assert statements. When -O is used, *all* *bytecode* is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode.
- Passing two -O flags to the Python interpreter (-OO) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only __doc__ strings are removed from the bytecode, resulting in more compact .pyo files. Since some programs may rely on having these available, you should only use this option if you know what you're doing.
- A program doesn't run any faster when it is read from a .pyc or .pyo file than when it is read from a .py file; the only thing that's faster about .pyc or .pyo files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a .pyc or .pyo file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a .pyc or .pyo file directly on the command line.
- It is possible to have a file called spam.pyc (or spam.pyo when -O is used) without a file spam.py for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
- The module compileall can create .pyc files (or .pyo files when -O is used) for all modules in a directory.

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__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
 '_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
 'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
 '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__', '__package__', 'a', 'fib', 'fibo', 'sys']
```

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

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', '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, item 은 패키지의 서브 모듈 (또는 서브 패키지) 일 수도 있고 함수, 클래스, 변수 등 패키지에 정의된 다른 이름들일 수도 있음에 유의하세요. `import` 문은 먼저 item 이 패키지에 정의되어 있는지 검사하고, 그렇지 않으면 모듈이라고 가정하고 로드를 시도합니다. 찾지 못한다면, `ImportError` 예외를 일으킵니다.

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

6.4.1 패키지에서 * 임포트 하기

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

유일한 해결책은 패키지 저자가 패키지의 색인을 명시적으로 제공하는 것입니다. `import` 문은 다음과 같은 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.

Remember, there is nothing wrong with using `from package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

6.4.2 패키지 내부 간의 참조

The submodules often need to refer to each other. For example, the `surround` module might use the `echo` module. In fact, such references are so common that the `import` statement first looks in the containing package before looking in the standard module search path. Thus, the `surround` module can simply use `import echo` or `from echo import echofilter`. If the imported module is not found in the current package (the package of which the current module is a submodule), the `import` statement looks for a top-level module with the given name.

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.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the `from module import name` form of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

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

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application should always use absolute imports.

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

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

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string types have some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the `str.format()` method.

`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.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

몇 가지 예를 듭니다:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'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),
...     # Note trailing comma 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
```

(Note that in the first example, one space between each column was added by the way `print` works: by default it adds spaces between its arguments.)

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!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets refers to the position of the object passed into the `str.format()` method.

```
>>> 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.
```

`'!s'` (apply `str()`) and `'!r'` (apply `repr()`) can be used to convert the value before it is formatted.

```
>>> import math
>>> print 'The value of PI is approximately {}.'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}.'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

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.
```

More information can be found in the string-formatting section.

7.2 파일을 읽고 쓰기

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

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.

On Windows, `'b'` appended to the mode opens the file in binary mode, so there are also modes like `'rb'`, `'wb'`, and `'r+b'`. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading

and writing such files. On Unix, it doesn't hurt to append a `'b'` to the mode, so you can use it platform-independently for all binary files.

7.2.1 파일 객체의 매소드

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

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

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

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> 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,

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)` writes the contents of *string* to the file, returning `None`.

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

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

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

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

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

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

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

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"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a file. So if `f` is a *file object* opened for writing, we can do this:

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

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

```
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
```

The parser repeats the offending line and displays a little 〈arrow〉 pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the keyword `print`, since a colon (`':'`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

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: integer division or modulo 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: cannot concatenate 'str' and 'int' objects
```

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(raw_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
```

Note that the parentheses around this tuple are required, because `except ValueError, e:` was the syntax used for what is normally written as `except ValueError as e:` in modern Python (described below). The old syntax is still supported for backwards compatibility. This means `except RuntimeError, TypeError` is not equivalent to `except (RuntimeError, TypeError):` but to `except RuntimeError as TypeError:` which is not what you want.

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 IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
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 IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

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

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

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`.

One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> 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
...     x, y = inst.args
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has an argument, it is printed as the last part (〈detail〉) of the message for unhandled exceptions.

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 detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo 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
```

The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

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

Programs may name their own exceptions by creating a new exception class (see 클래스 for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly. For example:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: 'oops!'
```

In this example, the default `__init__()` of `Exception` has been overridden. The new behavior simply creates the *value* attribute. This replaces the default behavior of creating the *args* attribute.

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:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

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

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg
```

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>
```

A *finally clause* is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed《on the way out》when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example (having `except` and `finally` clauses in the same `try` statement works as of Python 2.5):

```
>>> 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
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,
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

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

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.

CHAPTER 9

클래스

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 *Private Variables and Class-local References* 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.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `__builtin__`.)

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.

스코프 는 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.

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such

[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.

a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

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.)

A special quirk of Python is that – if no `global` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)

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

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

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…

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

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.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is `False` since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).

9.5.1 다중 상속

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

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

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first — searching `Base2` and `Base3` before the base classes of `Base1` — looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

For *new-style classes*, the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see https://www.python.org/download/releases/2.3/mro/.

9.6 Private Variables and Class-local References

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.

Notice that code passed to `exec`, `eval()` or `execfile()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

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.

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

9.8 Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

```
raise instance.__class__, instance
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

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

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.

9.9 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,
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    it.next()
StopIteration
```

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

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

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

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

9.10 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
```

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

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

9.11 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 = dict((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:\\Python26'
>>> 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')
>>> shutil.move('/build/executables', '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)
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(xrange(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
```

10.7 인터넷 액세스

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are urllib2 for retrieving data from URLs and smtplib for sending mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     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
...
... 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

Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, zipfile and tarfile.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'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, 0.0) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

unittest 모듈은 doctest 모듈만큼 쉬운 것은 아니지만, 더욱 포괄적인 테스트 집합을 별도의 파일로 관리할 수 있게 합니다:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests
```

10.12 배터리가 포함됩니다

파이썬은《배터리가 포함됩니다》철학을 갖고 있습니다. 이는 더 큰 패키지의 정교하고 강력한 기능을 통해 가장 잘 나타납니다. 예를 들어:

- The xmlrpclib and SimpleXMLRPCServer modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- email 패키지는 MIME 및 기타 RFC 2822 기반 메시지 문서를 포함하는 전자 메일 메시지를 관리하기 위한 라이브러리입니다. 실제로 메시지를 보내고 받는 smtplib 와 poplib 와는 달리, email 패키지는 복잡한 메시지 구조 (첨부 파일 포함) 를 작성하거나 해독하고 인터넷 인코딩과 헤더 프로토콜을 구현하기 위한 완벽한 도구 상자를 가지고 있습니다.

- The `xml.dom` and `xml.sax` packages provide robust support for parsing this popular data interchange format. Likewise, the `csv` module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- 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

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

`pprint` 모듈은 인터프리터가 읽을 수 있는 방식으로 내장 객체나 사용자 정의 객체를 인쇄하는 것을 보다 정교하게 제어할 수 있게 합니다. 결과가 한 줄보다 길면 《예쁜 프린터》가 줄 바꿈과 들여쓰기를 추가하여 데이터 구조를 보다 명확하게 나타냅니다:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...       'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

`textwrap` 모듈은 텍스트의 문단을 주어진 화면 너비에 맞게 포맷합니다:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> 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 누락된 경우 자리표시자를 변경하지 않습니다:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template 서브 클래스는 사용자 정의 구분자를 지정할 수 있습니다. 예를 들어 사진 브라우저를 위한 일괄 이름 바꾸기 유틸리티는 현재 날짜, 이미지 시퀀스 번호 또는 파일 형식과 같은 자리표시자에 백분율 기호를 사용하도록 선택할 수 있습니다:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_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

data = open('myfile.zip', 'rb').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.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `Queue` module to feed that thread with requests from other threads. Applications using `Queue.Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

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:/python26/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 *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
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

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.

- install-index explains how to install external modules written by other Python users.

- 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.

For Python-related questions and problem reports, you can post to the newsgroup `comp.lang.python`, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check

the list of Frequently Asked Questions (also called the FAQ). Mailing list archives are available at https://mail.python.org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

CHAPTER 13

대화형 입력 편집 및 히스토리 치환

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the Unix and Cygwin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

13.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: `C-A` (Control-A) moves the cursor to the beginning of the line, `C-E` to the end, `C-B` moves it one position to the left, `C-F` to the right. Backspace erases the character to the left of the cursor, `C-D` the character to its right. `C-K` kills (erases) the rest of the line to the right of the cursor, `C-Y` yanks back the last killed string. `C-underscore` undoes the last change you made; it can be repeated for cumulative effect.

13.2 History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. `C-P` moves one line up (back) in the history buffer, `C-N` moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the `Return` key passes the current line to the interpreter. `C-R` starts an incremental reverse search; `C-S` starts a forward search.

13.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called ~/.inputrc. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for Tab in Python is to insert a Tab character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your ~/.inputrc. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using Tab for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:[1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the Tab key to the completion function, so hitting the Tab key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as string.a, it will evaluate the expression up to the final '.' and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a __getattr__() method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as os, which turn out to be needed in most sessions with the interpreter.

[1] Python will execute the contents of a file identified by the PYTHONSTARTUP environment variable when you start an interactive interpreter. To customize Python even for non-interactive mode, see 커스터마이제이션 모듈.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4 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 14

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...
```

Stop at any finite number of bits, and you get an approximation.

On a typical machine running Python, there are 53 bits of precision available for a Python float, so the value stored internally when you enter the decimal number `0.1` is the binary fraction

```
0.0001100110011001100110011001100110011001100110011001100110011010
```

which is close to, but not exactly equal to, 1/10.

It's easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

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

```
>>> 0.1
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you're simply rounding the *display* of the true machine value. This fact becomes apparent as soon as you try to do arithmetic with these values

```
>>> 0.1 + 0.2
0.30000000000000004
```

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).

Other surprises follow from this one. For example, if you try to round the value 2.675 to two decimal places, you get this

```
>>> round(2.675, 2)
2.67
```

The documentation for the built-in `round()` function says that it rounds to the nearest value, rounding ties away from zero. Since the decimal fraction 2.675 is exactly halfway between 2.67 and 2.68, you might expect the result here to be (a binary approximation to) 2.68. It's not, because when the decimal string `2.675` is converted to a binary floating-point number, it's again replaced with a binary approximation, whose exact value is

```
2.674999999999999822364316059974953532218933105468 75
```

Since this approximation is slightly closer to 2.67 than to 2.68, it's rounded down.

If you're in a situation where you care which way your decimal halfway-cases are rounded, you should consider using the `decimal` module. Incidentally, the `decimal` module also provides a nice way to 《see》 the exact value that's stored in any particular Python float

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.674999999999999822364316059974953532218933105468 75')
```

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

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.

As that says near the end, 《there are no easy answers.》 Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. For fine control over how a float is displayed see the `str.format()` method's format specifiers in formatstrings.

14.1 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.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Why is that? 1/10 and 2/10 are not exactly representable as a binary fraction. Almost all machines today (July 2010) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 《double precision》. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2**_N_ where *J* is an integer containing exactly 53 bits. Rewriting

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

를

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

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

```
>>> 2**52
4503599627370496
>>> 2**53
9007199254740992
>>> 2**56/10
7205759403792793
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

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

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

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

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or

```
7205759403792794 / 72057594037927936
```

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:

```
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794 * 10**30 // 2**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. In versions prior to Python 2.7 and Python 3.1, Python rounded this value to 17 significant digits, giving 〈0.10000000000000001〉. In current versions, Python displays a value based on the shortest decimal fraction that rounds correctly back to the true binary value, resulting simply in 〈0.1〉.

CHAPTER 15

부록

15.1 대화형 모드

15.1.1 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.

15.1.2 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 python
```

(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.

15.1.3 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)
```

15.1.4 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/python2.7/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) Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

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

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

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

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

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

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

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

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

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

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

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

바이트 코드 명령어들의 목록은 dis 모듈 도큐멘테이션에 나온다.

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

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

coercion (코어션) The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

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

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

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

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

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

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

@staticmethod
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def f(...):
    ...
```

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

descriptor (디스크립터) Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

디스크립터의 메서드들에 대한 자세한 내용은 descriptors 에 나온다.

dictionary (딕셔너리) An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary view (딕셔너리 뷰) The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

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

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

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

expression (표현식) A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

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

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

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

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

finder (파인더) An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See **PEP 302** for details.

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

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

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

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

garbage collection (가비지 수거) The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

generator (제너레이터) A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

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

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

GIL 전역 인터프리터 록 을 보세요.

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

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

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

hashable (해시 가능) An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

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

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

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

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

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

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

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

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

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

iterable (이터러블) An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ⋯). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator (이터레이터) An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

typeiter 에 더 자세한 내용이 있다.

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

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument (키워드 인자) 인자 를 보세요.

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

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

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

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

list comprehension (리스트 컴프리헨션) A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader (로더) An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

magic method An informal synonym for *special method*.

mapping (매핑) A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

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

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.

MRO 메서드 결정 순서 를 보세요.

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

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

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

namespace (이름 공간) The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

nested scope (중첩된 스코프) The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class (뉴스타일 클래스) Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in newstyle.

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

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

parameter (파라미터) A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

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

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

- 위치-전용 *(positional-only)*: 위치로만 제공될 수 있는 인자를 지정한다. 파이썬은 위치-전용 파라미터를 정의하는 문법을 갖고 있지 않다. 하지만, 어떤 매장 함수들은 위치-전용 파라미터를 갖는다 (예를 들어, `abs()`).
- 가변-위치 *(var-positional)*: (다른 파라미터들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정한다. 이런 파라미터는 파라미터 이름에 `*` 를 앞에 붙여서 정의될 수 있다, 예를 들어 다음에서 *args*:

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

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

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

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

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

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

PEP 1 참조하세요.

positional argument (위치 인자) 인자 를 보세요.

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

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

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

더 깔끔한, 파이썬다운 방법은 이렇다:

```
for piece in food:
    print piece
```

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

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence (시퀀스) An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

slice (슬라이스) An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

special method (특수 메서드) 파이썬이 형에 어떤 연산을, 덧셈 같은, 실행할 때 묵시적으로 호출되는 메서드. 이런 메서드는 두 개의 밑줄로 시작하고 끝나는 이름을 갖고 있다. 특수 메서드는 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 similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

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

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

universal newlines (유니버설 줄 넘김) A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `str.splitlines()` for an additional use.

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

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

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

APPENDIX B

About these documents

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

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

Many thanks go to:

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

B.1 Contributors to the Python Documentation

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

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

APPENDIX C

History and License

C.1 History of the software

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

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

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

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

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

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

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

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

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.18

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

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 2.7.18 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   2.7.18.

4. PSF is making Python 2.7.18 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

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

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

8. By copying, installing or otherwise using Python 2.7.18, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

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

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

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

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

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

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

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

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

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

C.3 Licenses and Acknowledgements for Incorporated Software

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

C.3.1 Mersenne Twister

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

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

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

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

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


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

C.3.2 Sockets

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.3 Floating point exception control

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

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

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

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

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

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Asynchronous socket services

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

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

C.3.6 Cookie management

The `Cookie` module contains the following notice:

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

               All Rights Reserved

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

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

C.3.7 Execution tracing

The `trace` module contains the following notice:

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

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

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

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

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


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

C.3.8 UUencode and UUdecode functions

The `uu` module contains the following notice:

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.9 XML Remote Procedure Calls

The `xmlrpclib` module contains the following notice:

```
    The XML-RPC client interface is

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

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

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

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

C.3.10 test_epoll

The `test_epoll` contains the following notice:

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

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Select kqueue

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

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

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

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

C.3.12 strtod and dtoa

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.13 OpenSSL

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

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

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */
```

C.3.14 expat

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

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

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

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

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

C.3.15 libffi

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

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

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

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

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

C.3.16 zlib

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

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

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

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

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

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

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

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

APPENDIX D

저작권

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

색인

Non-alphabetical

... , **107**
*
    글, 26
**
    글, 27
2to3, **107**
>>>, **107**
__all__, 48
__builtin__
    모듈, 46
__future__, **110**
__slots__, **114**
객체
    file, 54
    method, 71
글
    *, 26
    **, 27
    for, 20

A

abstract base class (추상 베이스 클래스), **107**
argument (인자), **107**
attribute (어트리뷰트), **108**

B

BDFL, **108**
bytecode (바이트 코드), **108**
bytes-like object (바이트열류 객체), **108**

C

class (클래스), **108**
classic class, **108**
coding
    style, 28
coercion (코어션), **108**
compileall
    모듈, 44
complex number (복소수), **108**
context manager (컨텍스트 관리자), **108**
CPython, **108**

D

decorator (데코레이터), **108**
descriptor (디스크립터), **109**
dictionary (딕셔너리), **109**
dictionary view (딕셔너리 뷰), **109**
docstring (독스트링), **109**
docstrings, 22, 27
documentation strings, 22, 27
duck-typing (덕 타이핑), **109**

E

EAFP, **109**
expression (표현식), **109**
extension module (확장 모듈), **109**

F

file
    객체, 54
file object (파일 객체), **109**
file-like object (파일류 객체), **109**
finder (파인더), **110**
floor division (정수 나눗셈), **110**
for
    글, 20
function (함수), **110**

G

garbage collection (가비지 수거), **110**
generator, 110
generator (제너레이터), **110**
generator expression, 110
generator expression (제너레이터 표현식), **110**
GIL, **110**
global interpreter lock (전역 인터프리터 록), **110**

H

hashable (해시 가능), **110**
help
    내장 함수, 81

I

IDLE, **111**
immutable (불변), **111**
importer (임포터), **111**
importing (임포팅), **111**
integer division, **111**
interactive (대화형), **111**
interpreted (인터프리티드), **111**
iterable (이터러블), **111**
iterator (이터레이터), **111**

J

json
    모듈, 56

K

key function (키 함수), **111**
keyword argument (키워드 인자), **112**

L

lambda (람다), **112**
LBYL, **112**
list (리스트), **112**
list comprehension (리스트 컴프리헨션), **112**
loader (로더), **112**

M

magic
    method, 112
magic method, **112**
mangling
    name, 76
mapping (매핑), **112**
metaclass (메타 클래스), **112**
method
    magic, 112
    special, 114
    객체, 71
method (메서드), **112**
method resolution order (메서드 결정 순서), **112**
module
    search path, 44
module (모듈), **112**
MRO, **113**
mutable (가변), **113**

N

name
    mangling, 76
named tuple (네임드 튜플), **113**
namespace (이름 공간), **113**
nested scope (중첩된 스코프), **113**
new-style class (뉴스타일 클래스), **113**

O

object (객체), **113**
open
    내장 함수, 54

P

package (패키지), **113**
parameter (파라미터), **113**
PATH, 44, 105
path
    module search, 44
PEP, **114**
positional argument (위치 인자), **114**
Python 3000 (파이썬 *3000*), **114**
Pythonic (파이썬다운), **114**
PYTHONPATH, 44, 45
PYTHONSTARTUP, 98, 106

R

readline
    모듈, 98
reference count (참조 횟수), **114**
rlcompleter
    모듈, 98

S

search
    path, module, 44
sequence (시퀀스), **114**
slice (슬라이스), **114**
special
    method, 114
special method (특수 메서드), **114**
statement (문장), **114**
strings, documentation, 22, 27
struct sequence (구조체 시퀀스), **114**
style
    coding, 28
sys
    모듈, 45

T

triple-quoted string (삼중 따옴표 된 문자열), **115**
type (형), **115**

U

unicode

내장 함수, 15
universal newlines (유니버설 줄 넘김), **115**

V

virtual environment (가상 환경), **115**
virtual machine (가상 기계), **115**

X

내장 함수
help, 81
open, 54
unicode, 15
모듈
__builtin__, 46
compileall, 44
json, 56
readline, 98
rlcompleter, 98
sys, 45

Y

파이썬 향상 제안
PEP 1, 114
PEP 8, 28
PEP 238, 110
PEP 278, 115
PEP 302, 110, 112
PEP 343, 108
PEP 3116, 115
환경 변수
PATH, 44, 105
PYTHONPATH, 44, 45
PYTHONSTARTUP, 98, 106

Z

Zen of Python (파이썬 젠), **115**