PART

IX

부록

APPENDIX

A

파이썬 설치와 환경 설정

이 부록에서는 파이썬 설치와 환경 설정에 대한 상세에 대하여 설명하겠다. 이 주제가 새로운 사람들에게 유용한 자료가 될 것이다. 이 내용에 대하여 여기에서 다루는 이유는 모든 독자가 이 주제를 직접 다룰 필요가 있는 것은 아니기 때문이다. 여기에서는 환경 변수나 명령 라인 인수와 같은 주변 주제들을 다루기 때문에 이 내용은 대부분의 독자들이 잠깐이라도 훑어보는 것이 좋을 것이다.

파이썬 인터프리터 설치하기

파이썬 스크립트를 실행하기 위해서는 파이썬 인터프리터가 필요하기 때문에 파이썬을 사용하는 첫 단계는 대체로 파이썬을 설치하는 일이 된다. 여러분 컴퓨터에 이미 파이썬이 설치되어 있지 않다면, 여러분 컴퓨터에 파이썬 최신 버전을 받아서 설치하고, 아마도 설정해야 할 것이다. 컴퓨터당 한 번씩만 이 작업을 수행하고, 만약 2장에서 설명한 프로즌 바이너리나 자가 설치 시스템을 실행한다면, 설치 작업은 거의 미미한 일이 될 것이다.

파이썬이 이미 설치되어 있는가?

가장 먼저 할 일은 여러분의 컴퓨터에 이미 최신 파이썬이 설치되어 있는지 여부를 확인하는 것이다. 만약 리눅스, 맥 OS X 또는 일부 유닉스 시스템에서 작업한다면 아마 여러분 컴퓨터

에 파이썬은 이미 설치되어 있을 수도 있다. 가장 최신 버전에서 한두 버전 정도 아래 버전 일 수는 있지만 말이다. 여기 이를 확인하는 방법에 대해 알아보자.

- **윈도우 7**이나 그 이전 버전에서는 시작 버튼을 눌러서 모든 프로그램 메뉴(화면 좌측 아래)에 파이썬이 있는지 확인하면 된다.[1]
- **맥 OS X**에서는 터미널 창(응용 프로그램 ➡ 유틸리티 ➡ 터미널)을 열고 프롬프트에 python을 입력하면 된다. 파이썬, IDLE, tkinter GUI 툴킷은 이 시스템의 표준 컴포넌트다
- **리눅스와 유닉스**에서는 셸 프롬프트(터미널 창)에서 python을 써 넣고 무슨 일이 일어나는지 확인하면 된다. 그렇지 않으면, 기본 설치 장소(/usr/bin, /usr/local/bin 등)에서 'python'을 찾아보아도 될 것이다. 맥과 동일하게 파이썬은 리눅스 시스템의 표준 구성 요소다.

파이썬이 이미 설치되어 있다면 최신 버전인지를 확인하자. 최근의 파이썬 버전이면 이 책을 학습하기에 충분하겠지만, 이번 판은 특히 파이썬 3.3과 2.7을 중심으로 기술했으므로 이 책의 예제를 실행하려면 둘 중에 하나는 설치하는 것이 좋다.

버전에 대해 말하자면 서문에서 다룬대로 파이썬을 새로 배우고 기존 2.X 코드를 다룰 필요가 없다면, 파이썬 3.3 또는 그 이후 버전으로 시작하기를 권한다. 만약 그렇지 않다면 일반적으로 파이썬 2.7을 사용해야 할 것이다. 일부 유명한 파이썬 기반의 시스템은 여전히 예전 릴리즈를 사용하기도 한다(2.6, 그리고 심지어 2.5도 여전히 보편적으로 사용되고 있다). 따라서 만약 기존 시스템으로 작업해야 한다면, 수요에 맞는 버전을 확인하고 사용하여야 한다. 다음 절에서 다양한 파이썬 버전을 어디에서 받을 수 있는지에 대하여 설명한다.

어디서 파이썬을 가져올 것인가?

만약 여러분 컴퓨터에 파이썬이 없다면, 파이썬을 설치해야 할 것이다. 반가운 소식은 파이썬은 오픈 소스 시스템이어서 웹에서 무료로 구할 수 있으며, 대부분의 플랫폼에서 쉽게 설치된다는 것이다.

가장 최신, 최고의 **표준 파이썬** 릴리즈는 파이썬 공식 웹 사이트인 **https://www.python.org**에서 항상 받을 수 있다. 그 웹 페이지에서 Downloads 링크를 찾아서 작업할 플랫폼의 릴리즈

1 옮긴이 윈도우 10에서는 윈도우 8에서 없어졌던 시작 버튼이 복원되었으므로 윈도우 7과 같은 방법을 사용한다.

를 선택한다. 윈도우용 자동 실행 파일(설치를 위해 실행만 시키면 됨), 맥 OS X을 위한 인스톨러 디스크 이미지(맥 규격에 따라 설치됨), 소스 코드 배포판(보통 리눅스, 유닉스, 또는 OS X 머신에서 컴파일되어 인터프리터를 생성함) 등을 발견할 수 있을 것이다.

요즘에는 리눅스에서 파이썬이 표준이지만, 웹에서 리눅스를 위한 RPM을 받을 수도 있다(rpm을 활용하여 푼다). 파이썬 웹 사이트는 다른 플랫폼을 위한 버전을 관리하는 페이지로의 링크도(https://www.python.org) 내부 또는 외부 사이트 포함) 제공하고 있다. 예를 들어, 구글 안드로이드를 위한 제3자 파이썬 설치파일이나 애플 iOS에 파이썬을 설치하는 앱을 찾을 수 있다.

구글 웹 검색은 파이썬 설치 패키지를 찾는 또 다른 훌륭한 방법이다. 다른 플랫폼 중 아이팟(iPod), 팜 PC, 노키아 휴대폰, 플레이스테이션, PSP, 솔라리스(Solaris), AS/400, 윈도우 모바일 등을 위한 딕셔너리에 빌드된 파이썬을 찾을 수 있다. 이 중 몇몇은 제품 커브가 하향세로 들어서면서 릴리즈가 드물긴 하겠지만 말이다.

만약 윈도우 머신에 유닉스 환경을 구축하고자 한다면, Cygwin을 설치하고서 Cygwin용 파이썬을 설치할 수도 있다(http://www.cygwin.com 참조). Cygwin은 GPL 라이선스의 라이브러리와 도구로 윈도우 머신에서 모든 유닉스 기능을 제공하며, 제공하는 모든 유닉스 도구를 사용하는 딕셔너리에 빌드된 파이썬을 포함하고 있다.

파이썬은 리눅스 배포판이나 일부 제품, 컴퓨터 시스템 또는 몇몇 다른 파이썬 책의 부록으로 제공되는 CD-ROM 등에서 찾아볼 수 있다. 이는 현재 릴리즈보다 이전 버전일 경우가 많으나, 그리 심하게 오래되지는 않았을 것이다.

또한, 파이썬은 일부 무료 또는 상업용 개발 도구에서도 찾아볼 수 있다. 이 글을 쓰는 시점에는 다음과 같은 배포판이 존재한다.

ActiveState ActivePython[2]

파이썬과 과학 컴퓨팅, 윈도우 또는 다른 개발 요건을 위한 확장 프로그램과 결합한 패키지로 PyWin32, PythonWin IDE를 포함하고 있다.

Enthought Python Distribution[3]

파이썬과 과학 컴퓨팅 요건에 맞춘 다수의 추가 라이브러리와 도구들의 결합물이다.

---

2 옮긴이 ActivePython은 2018년 1월 현재 2.7.14, 3.5.4, 3.6.0 버전을 다운로드할 수 있다.

3 옮긴이 Enthought Python Distribution은 2018년 1월 현재 1.9.0 버전을 다운로드할 수 있다.

Portable Python[4]

휴대용 기기에서 직접 실행이 가능하도록 구성된 파이썬과 추가 패키지의 결합물이다.

Pythonxy[5]

Qt와 Spyder 기반의 과학 컴퓨팅을 지향하는 파이썬이다.

Conceptive Python SDK[6]

기업용, 데스크톱용, 데이터베이스용 애플리케이션을 목적으로 하는 번들이다.

PyIMSL Studio[7]

수치 해석을 위한 상업용 배포판이다.

Anaconda Python[8]

대용량 데이터 셋을 분석하고 시각화하기 위한 배포판이다.

이 내용은 언제든 변경될 여지가 있으므로, 위에 언급한 내용의 상세나 다른 배포판에 대해서는 웹에서 검색해 보는 것이 좋다. 이 중 일부는 무료지만 아닌 것들도 있으며, 무료와 유료 버전 둘을 모두 가지고 있는 것들도 있다. 앞에서 나열한 모든 배포판들은 https://www.python.org에서 무료로 다운로드할 수 있는 표준 파이썬에 추가 도구들을 조합한 것이므로 도구들을 추가 설치하는 절차를 단순화할 수 있다.

마지막으로 만약 대안적인 파이썬 구현에 관심이 있다면, 웹에서 Jython(파이썬을 자바 환경에 포팅)[9]과 IronPython(C#/.NET을 위한 파이썬)[10]에 대하여 검색해 보기 바란다. 이 둘에 대하여 2장에서 서술한 바 있다. 이들 시스템의 설치는 이 책에서는 논외로 한다.

설치 단계

일단 파이썬을 다운로드했으면 설치를 해야 한다. 설치 단계는 플랫폼에 따라 매우 다르기 때문에 주요 파이썬 플랫폼에 대한 몇 가지 주의 사항을 여기에 정리하였다(윈도우 위주로 소개하는 이유는 대부분의 파이썬 입문자들이 처음으로 언어를 접하는 플랫폼이기 때문이다).

---

4 옮긴이 Portable Python은 2018년 1월 현재 개발이 중단되었다.

5 옮긴이 Python(x,y)는 2015년 이후로 업데이트가 되지 않고 있으며, 2.7.10.0 이 가장 최신 버전이다.

6 옮긴이 Conceptive Python SDK는 2018년 1월 현재 웹 사이트에서 구매가 가능하지만, python 2.7.2 버전만을 지원한다.

7 옮긴이 pyIMSL Studio는 2018년 1월 현재 웹 사이트에서 몇 가지 인증 절차를 거쳐 Python Wrapper를 다운로드할 수 있다.

8 옮긴이 Anaconda Python은 2018년 1월 현재 python 3.6.4에 대응하는 버전을 다운로드할 수 있다.

9 옮긴이 Jython은 2015년에 2.7.0이 릴리즈된 이후 업데이트되지 않고 있다.

10 옮긴이 IronPython은 2016년 12월 이후 릴리즈되고 있지 않으며, 2.7.7이 가장 최신 버전이다.

윈도우

윈도우(XP, 비스타, 7, 8, 10을 포함하여)의 경우, 파이썬은 **자동 실행기**인 MSI 프로그램 파일로 제공되기 때문에 단순히 파일 아이콘을 더블 클릭하고, 기본 설치를 위해서는 모든 프롬프트에 '예'와 '다음'을 클릭하면 된다. 기본 설치에는 파이썬의 다큐멘테이션 세트가 포함되어 있고 tkinter(파이썬 2.X에서는 Tkinter) GUI, shelve 데이터베이스, IDLE 개발 GUI를 지원한다. 파이썬 3.3과 2.7은 보통 C:\Python33과 C:\Python27 디렉터리에 설치되며[11], 다른 디렉터리에 설치하기 원한다면 설치 시 변경할 수 있다.

편의성을 위해 윈도우 7과 이전 버전에서 파이썬은 설치 후 시작 => 모든 프로그램 메뉴에서 확인이 가능하다. 파이썬 메뉴에는 보편적인 작업에 빠르게 접근할 수 있도록 다섯 개의 항목이 있다. IDLE 사용자 인터페이스 시작하기, 모듈 다큐멘테이션 읽기, 대화형 세션 시작하기, 파이썬 표준 매뉴얼 읽기, 프로그램 제거하기가 그것이다. 이 중 대부분은 이 글의 다른 곳에서 세부적으로 다룰 개념과 관련이 있다.

윈도우에 설치되면 파이썬은 자동적으로 **파일명 연결**을 사용하여 자신을 파이썬 파일 아이콘을 클릭하면 해당 파일을 여는 프로그램으로 등록한다(프로그램 런치 기법은 3장에서 설명하였다). 윈도우에서 파이썬 소스 코드로 파이썬을 빌드할 수도 있지만, 보통은 이 방법을 사용하지 않기 때문에 여기에서는 다루지 않는다(python.org 참조).

윈도우 사용자를 위한 설치 관련 추가 내용으로는 세 가지가 있다. 첫째, 3.3에 탑재된 새로운 **윈도우 런처**(Windows launcher)에 대하여 소개하는 다음 부록을 반드시 확인해야 한다. 설치, 파일 연계, 명령 라인 관련 몇 가지 규칙이 바뀌지만, 만약 한 컴퓨터에 여러 파이썬 버전을 가지고 있다면(일례로 2.X와 3.X를 모두 가지고 있다면) 자산이 될 수 있다. 부록 B에 의하면 파이썬 3.3의 MSI 설치 파일 또한 PATH 변수에 파이썬 디렉터리를 포함하도록 설정하는 옵션을 가지고 있다.

두 번째로, **윈도우 비스타** 사용자는 보안 관련 특성으로 인해 설치상의 이슈에 맞닥뜨릴 수 있다. 이는 시간이 지나면서 해결이 된 듯하지만(비스타는 요새 상대적으로 사용하는 사람이 드물다), 만약 MSI 설치 파일을 직접 실행한다면 역시나 제대로 실행되지 않을 것이다. 이는 MSI 파일이 실제로 실행 가능한 형태가 아니고 정확하게 관리자 권한을 상속받은 것이 아니기 때문이다(레지스트리로 실행된다). 이 문제를 해결하기 위해서 적절한 권한을 가

11 옮긴이 이 책이 집필될 시점에 파이썬의 최신 버전은 3.3이었으나, 2018년 1월 현재 최신 버전은 3.6이다. 만약 독자가 설치한 버전이 python 3.6이라면, 이 경로는 C:\python36이 된다. 이 책에서는 원서의 표기에 따라 그대로 C:\python33으로 표기하지만, 독자가 설치한 버전에 따라 알맞은 경로로 입력하도록 하자.

지고 명령 라인에서 설치 파일을 실행하면 된다. 명령 프롬프트를 선택하고, '관리자 권한으로 실행(Run as administrator)'을 선택하고, 파이썬 MSI 파일이 위치한 디렉터리로 이동(cd)한 뒤, MSI 설치 파일에 다음과 같은 명령 라인을 이용하여 실행하면 된다. msiexec /i python-2.5.1.msi 명령을 실행하여 설치할 수 있다.

리눅스

리눅스의 경우 파이썬 또는 여러분이 원하는 다른 변형된 도구가 이미 존재하지 않는다면, 아마도 그에 대한 하나 또는 그 이상의 RPM 파일을 얻을 수 있다. RPM 파일은 일반적인 방식(자세한 내용은 RPM manpage를 참조할 것)으로 풀면 된다. 어떤 RPM을 다운로드했는지에 따라 파이썬 자체만 들어 있거나, tkinter GUI와 IDLE 환경을 추가적으로 지원하는 도구가 포함되어 있기도 하다. 리눅스는 유닉스와 유사한 시스템이므로 다음의 유닉스 관련 내용도 동일하게 적용할 수 있다.

유닉스

유닉스 시스템에서 파이썬은 보통 완전한 C 소스 코드 배포판을 컴파일하여 설치한다. 이는 일반적으로 파일을 풀고, 간단하게 config와 make 명령어를 실행시키면 된다. 파이썬은 자동적으로 컴파일될 시스템에 따라, 자신의 빌드 프로시저를 설정한다. 그러나 이 절차의 더 자세한 내용은 패키지의 README 파일을 참조하는 것이 좋다. 파이썬은 오픈 소스이므로 소스 코드는 무료로 배포하고 사용할 수 있을 것이다.

다른 플랫폼에서는 설치 관련 세부 작업이 크게 다를 수 있다. 하지만 보통은 각 플랫폼이 가지고 있는 보편적인 규칙에 따른다. 예를 들어, 팜 OS를 위하여 파이썬을 포팅한 'Pippy'를 설치하기 위해서는 여러분의 PDA와 호스트 동기화(hostsync) 작업을 해야 한다. Sharp Zaurus 리눅스 기반의 PDA를 위한 파이썬에서는 하나 또는 그 이상의 .ipk 파일이 필요하다. 설치를 위해서는 단순히 이 파일을 실행시키면 된다(여전히 설치 가능하겠지만, 요새는 이들 단말기를 찾는 것 자체가 도전적인 일일 것이다).

좀 더 최근에는 안드로이드와 iOS 플랫폼에서도 파이썬을 설치하고 사용할 수 있다. 하지만 설치와 활용 기법이 플랫폼에 지나치게 특화되어 있어, 여기에서는 다루지 않기로 한다. 추가적인 설치 절차와 최신 포팅 버전에 대해서는 파이썬 웹 사이트나 웹 검색을 통하여 확인해 볼 수 있다.

파이썬 환경 설정하기

파이썬 설치가 끝났으면, 파이썬이 코드를 실행하는 방식에 영향을 줄 수 있는 몇 가지 시스템 환경 설정이 필요할 수도 있다(이제 막 언어를 배우기 시작하였다면, 이 절은 완전히 건너뛰어도 된다. 일반적으로 기본 프로그램을 위해서 별도의 시스템 설정이 필요하지는 않다).

일반적으로 파이썬 인터프리터 행위의 일부분은 환경 변수 세팅과 명령 라인 옵션으로 설정할 수 있다. 이 절에서 우리는 이 둘에 대하여 간단히 살펴볼 것이다. 하지만 여기에서 소개하는 주제에 대하여 더 자세히 알아보고자 한다면 다른 관련 문서를 찾아보는 것이 좋다.

파이썬 환경 변수

환경 변수(혹자는 셸 변수나 DOS 변수로 알려진)는 파이썬 외부에 존재하는 시스템 범위의 설정으로, 해당 컴퓨터에서 실행될 때마다 일어나는 인터프리터 행위를 변경하기 위해 사용될 수 있다. 파이썬은 몇 안되는 환경 변수 세팅을 인지하지만, 여기에서의 설명을 보장하기에는 이 중 일부만으로도 충분하다. 표 A-1는 주요 파이썬 관련 환경 변수 세팅을 요약한 것이다(다른 정보는 파이썬 관련 자료에서 찾아볼 수 있을 것이다).

표 A-1 주요 환경 변수

변수	역할
PATH(또는 path)	시스템 셸 검색 경로('python' 검색 용도)
PYTHONPATH	파이썬 모듈 검색 경로(임포트용)
PYTHONSTARTUP	파이썬 대화형 스타트업 파일 경로
TCL_LIBRARY, TK_LIBRARY	GUI 확장 변수(tkinter)
PY_PYTHON, PY_PYTHON3, PY_PYTHON2	윈도우 런처 기본값(부록 B 참조)

이 변수들은 사용하기 간단하지만 몇 가지 주의 사항이 있다.

PATH

PATH 설정은 운영체제가 실행 가능한 프로그램을 전체 디렉터리 경로 없이 작동시킬 때, 해당 프로그램을 검색하는 디렉터리들을 나열한 것이다. 이는 보통 파이썬 인터프리터(유닉스의 파이썬 프로그램 또는 윈도우의 python.exe 파일)가 위치한 디렉터리를 포함해야 한다.

만약 여러분이 파이썬이 위치한 디렉터리에서 작업하거나 명령 라인에서 파이썬의 전체 경로를 입력한다면, 이 변수를 설정할 필요가 전혀 없다. 예를 들어, 윈도우에서 파이썬 코드를 실행하기 전에 무조건 cd C:\Python33(파이썬 실행 파일이 위치한 곳이다. 3장에서 설명했지만, 여기에 당신이 작성한 코드를 저장하는 것은 일반적으로 바람직하지 않다)을 실행하거나, 또는 항상 python 대신 C:\Python33\python으로 명령어를 입력한다면 PATH는 신경 쓸 필요가 없는 설정 변수다.

PATH 설정은 대체로 명령 라인에서 프로그램을 시작하기 위한 용도임을 주목할 필요가 있다. 이 설정은 일반적으로 아이콘 클릭이나 IDE를 통해 런칭하는 경우에는 상관없다. 전자는 파일명 연계를 사용하고, 후자는 내장된 메커니즘을 사용하며, 이들은 일반적으로 이러한 설정 단계가 필요 없다. 파이썬 3.3 설치 과정에서 PATH 환경 변수를 자동으로 설정하는 방법에 대해서는 부록 B를 참조하자.

PYTHONPATH

PYTHONPATH 설정은 PATH와 유사한 역할을 한다. 파이썬 인터프리터는 PYTHONPATH 변수를 통해 프로그램에 **임포트**할 모듈 파일의 위치를 찾는다. 이 변수를 사용하게 되면, 플랫폼에 따라 다른 디렉터리명의 리스트를 유닉스에서는 콜론으로, 윈도우에서는 세미콜론으로 구분하여 설정한다. 이 리스트는 일반적으로 여러분의 소스 코드 디렉터리만 포함하고 있다. 이 내용은 sys.path의 모듈 임포트 검색 경로에 스크립트 컨테이너 디렉터리, .pth 경로 파일 설정, 그리고 표준 라이브러리 디렉터리와 함께 통합된다.

만약 **디렉터리 간 임포트**를 수행하지 않는다면, 이 변수를 세팅할 필요가 없다. 파이썬은 항상 프로그램의 최상위 파일의 홈 디렉터리를 자동 검색하기 때문이다. 이 설정은 오직 모듈이 다른 디렉터리에 위치한 모듈을 임포트할 필요가 있을 때만 필요하다. PYTHONPATH 대안으로서의 .pth 경로 파일에 대한 논의는 이 부록의 후반부에서 다룰 예정이다. 모듈 검색 경로에 대한 더 많은 내용은 22장을 참조하면 된다.

PYTHONSTARTUP

PYTHONSTARTUP 이 파이썬 코드의 파일 경로명을 설정하게 되면, 파이썬은 여러분이 대화형 인터프리터를 시작할 때마다 (마치 여러분이 대화형 명령 라인에 그 코드를 입력한 것처럼) 자동으로 해당 파일의 코드를 실행한다. 이는 드물게 사용되지만, 만약 여러분이 대화형 작업을 수행할 때 특정 유틸리티가 항상 로딩되어 있어야 한다면 이 설정은 매우 유용한 방법이다. 이는 파이썬 세션을 시작할 때마다 임포트하는 수고를 덜어 준다.

tkinter 세팅

만약 tkinter GUI 툴킷(2.X에서는 Tkinter)을 사용하고자 한다면, 표 A-1의 마지막 줄의 두 가지 GUI 변수에 Tcl과 Tk 시스템의 소스 라이브러리 디렉터리명을 설정해야 할 것이다(PYTHONPATH와 거의 유사). 하지만 이 설정은 윈도우 시스템에서는 tkinter가 파이썬과 함께 설치되기 때문에 필요 없다. 그리고 맥 OS X과 리눅스 시스템에서는 기반 Tcl과 Tk 라이브러리가 인식되지 않거나, 비표준 디렉터리에 위치하지 않는 한 필요 없다(자세한 내용은 python.org의 Download 페이지 참조).

PY_PYTHON, PY_PYTHON3, PY_PYTHON2

이 설정들은 여러분이 새로운 파이썬 3.3(이 책 집필 당시)에 탑재되어 있고 또는 다른 버전 용도로 별도로도 사용 가능한 윈도우 런처를 사용할 때, 기본 파이썬을 정하기 위해 사용한다. 런처에 대해서는 부록 B에서 다룰 예정이므로 더 자세한 내용은 여기서 다루지 않는다.

이런 환경 설정은 파이썬 자체에 포함되어 있는 것이 아니기 때문에 이것을 언제 설정하는가는 크게 중요하지 않다. 이 환경 설정은 파이썬 설치 전이나 설치 후에 모두 설정할 수 있으며, 파이썬을 실제로 실행할 때 여러분이 원하는 값으로 설정되기만 하면 된다. 이러한 환경 설정이 적용되기 위해서는 설정 후에 파이썬 IDE나 대화형 세션을 반드시 재시작해야 한다.

리눅스와 맥에서의 Tkinter와 IDLE GUI

3장에서 설명한 IDLE 인터페이스는 파이썬 tkinter GUI 프로그램이다. tkinter 모듈(2.X에서의 이름은 Tkinter다)은 윈도우에서 표준 파이썬을 설치하면 자동으로 함께 설치되는 GUI 툴킷이며, 맥 OS X과 대부분의 리눅스 설치에 내재된 부분이다.

일부 **리눅스** 시스템에서는 기반 GUI 라이브러리가 표준 설치 컴포넌트가 아닐 수도 있다. 리눅스에서 파이썬에 GUI 지원을 추가하기 위해서는, sudo yum install tkinter를 명령 라인에서 실행하여 자동으로 tkinter의 기반 라이브러리를 설치하면 된다. 이 방식은 **yum** 설치 프로그램을 이용할 수 있는 리눅스 배포판(과 일부 다른 시스템)에서는 동작해야 한다. 다른 시스템 관련해서는 각자 플랫폼에 대한 설치 가이드를 참조하면 된다.[12]

아울러 3장에서 논의했듯이, **맥 OS X**에서 IDLE은 아마도 여러분의 **응용 프로그램** 폴더 아래의 **MacPython**(또는 **Python N.M**) 폴더에 있을 것이다. 그러나 만약 IDLE에 문제가 있다면, python.org의 다운로드 페이지를 반드시 확인해 보아야 한다. 일부 OS X 버전에서는 업데이트를 설치해야 할 수도 있다(3장 참조)

12 옮긴이 Ubuntu에서는 sudo apt-get install python3-tk 명령을 실행하여 설치할 수 있다.

설정 옵션 지정 방법

파이썬 관련 환경 변수를 설정하는 방법과 환경 변수에 무엇을 설정할 것인가는 여러분이 작업할 컴퓨터의 유형에 따라 달라진다. 다시 말하지만, 환경 설정을 당장 해야 할 필요는 없다는 것을 기억하자. 특히 여러분이 IDLE(3장 참조)에서 작업하고 모든 파일을 동일한 디렉터리에 저장한다면, 환경 설정은 아마 필요 없을 것이다.

하지만 설명을 위해 여러분 컴퓨터 어딘가에 utilities와 package1이라 불리는 디렉터리에 일반적으로 유용한 모듈 파일을 가지고 있고, 이 모듈들을 다른 디렉터리에 위치한 파일로부터 임포트하고자 한다고 가정해 보자. 말하자면 spam.py라는 파일을 utilities나 package1 디렉터리에서 로드한다는 것은, 임의 경로에 있는 임의 파일에서 다음과 같은 임포트 구문을 사용할 수 있다는 뜻이다.

```
import spam
```

이 문장이 제대로 동작하려면, 어떻게 해서든 모듈 검색 경로에 spam.py가 포함된 디렉터리를 포함되도록 설정해야 할 것이다. 여기에 일례로, PYTHONPATH를 활용하는 설정하는 방법에 대한 몇 가지 팁이 있다. 필요에 따라 PATH와 같은 다른 환경 설정에서도 동일하게 설정하면 된다(3.3에서는 PATH를 자동으로 설정할 수 있다. 부록 B 참조).

유닉스/리눅스 셸 변수

유닉스 시스템에서 환경 변수를 설정하는 방법은 여러분이 사용하는 셸에 따라 다르다. csh 셸을 사용한다면, 여러분의 .cshrc 또는 .login 파일에 다음과 같은 줄을 추가하여 파이썬의 모듈 검색 경로를 설정할 것이다.

```
setenv PYTHONPATH /usr/home/pycode/utilities:/usr/lib/pycode/package1
```

이는 파이썬에게 임포트될 모듈을 두 개의 사용자 정의 디렉터리에서 찾아보라고 말한다. 그 대신에 ksh를 사용한다면, 환경 설정을 위해 .kshrc 파일에 다음과 같은 줄이 등장할 것이다.

```
export PYTHONPATH="/usr/home/pycode/utilities:/usr/lib/pycode/package1"
```

다른 셸들은 다른(하지만 유사한) 구문을 사용할 것이다.

DOS 변수(와 구 윈도우 버전)

만약 여러분이 MS-DOS를 사용하거나 꽤 오래 전에 나온 윈도우 버전을 사용한다면, 환경 변수 설정 명령을 C:\autoexec.bat 파일에 추가하고 변경 사항이 반영될 수 있도록 컴퓨터를 리부팅해야 할 것이다. 이런 컴퓨터에서의 환경 설정 명령은 DOS 고유의 구문을 갖는다.

```
set PYTHONPATH=c:\pycode\utilities;d:\pycode\package1
```

이러한 명령어를 DOS 콘솔 창에서 타이핑할 수도 있지만, 이 경우 그 환경 설정은 해당 콘솔 창에 대해서만 유효하다. .bat 파일을 변경해야 이런 변경 사항이 영구적이며, 모든 프로그램에 공통으로 적용될 수 있다. 이런 기법은 근래에는 다음 절에서 다루는 기법에 의해 대체되고 있다.

윈도우 환경 변수 GUI

최근 윈도우 버전(XP, 비스타, 7, 8 등)에서는 PYTHONPATH 및 다른 변수에 대해 파일을 편집하고 명령 라인에 입력하고 리부팅하는 과정 대신에, 시스템 환경 변수 GUI를 통해 설정할 수 있다. 제어판을 선택하고(윈도우 7과 그 이전 버전에서 시작 버튼에 있음) 시스템 아이콘 선택, 고급 시스템 설정 탭이나 링크로 들어간 뒤, 하단의 환경 변수 버튼을 클릭하면 새로운 변수를 추가하거나 편집할 수 있다(PYTHONPATH은 보통 새로 사용자가 추가하는 변수임). 이전 절의 DOS set 명령어와 동일한 변수명과 값을 이용하여 설정하면 된다. 비스타에서는 모든 작업 단계별로 확인 작업을 거쳐야 할 수도 있다.

이 설정 과정을 마친 후, 컴퓨터를 재시작할 필요는 없으나, 변수 변경 작업을 파이썬이 열려 있는 상태에서 했다면 반드시 파이썬을 재시작해야 한다. 파이썬은 시작 시점에 한 번, 임포트 검색 경로를 설정하기 때문이다. 윈도우 명령 프롬프트 창을 사용한다면, 변경 사항 적용을 위해서는 동일하게 명령 프롬프트 창을 재시작해야 할 것이다.

윈도우 레지스트리

능숙한 윈도우 사용자라면 레지스트리 편집기를 이용하여 모듈 검색 경로를 설정할 수 있을지도 모른다. 이 도구를 열기 위해서 일부 윈도우에서는 regedit를 시작 ➡ 실행에서 입력한다. 윈도우 7과 10에서는 하단의 시작 버튼 내 메뉴 하단의 검색 필드에 입력하고, 윈도우 8에서는 명령 프롬프트 창에서 입력하면 된다. 일반적인 레지스트리 도구를 사용할 수 있는 컴퓨터

라면, 파이썬에 관련된 항목들을 찾아 변경할 수 있다. 만약 레지스트리를 다루는 데 서툰 사용자라면, 레지스트리를 직접 수정하는 일은 조심스럽고 에러가 발생할 가능성이 높은 작업이기 때문에 다른 방식을 사용하기를 권한다(이 일은 여러분 컴퓨터의 뇌를 수술하는 것과 마찬가지다. 조심해서 다루도록!).

경로 파일

모듈 검색 경로는 PYTHONPATH 변수 대신 .pth 경로(path) 파일로 확장할 수도 있는데, 이 경우에는 윈도우에서 text 파일에 다음과 같은 코드를 입력하고 경로 파일로 저장하면 된다(예 파일명 C\Python33\mypath.pth).

```
c:\pycode\utilities
d:\pycode\package1
```

내용은 플랫폼에 따라 다르며, 디렉터리는 플랫폼과 파이썬 버전에 따라 달라진다. 이 파일은 파이썬이 처음 시작할 때 자동으로 찾게 되어 있다.

경로 파일의 디렉터리명은 절대 경로로 작성하거나, 해당하는 파일이 위치한 디렉터리를 기준으로 한 상대 경로로 작성할 수 있다. 여러 개의 .pth 파일을 사용할 수도 있으며(그 파일들의 모든 디렉터리가 추가된다), .pth 파일은 자동으로 검증된 디렉터리라면 어디에서라도 등장할 수 있다. 이 디렉터리들은 플랫폼과 버전에 특화되어 있다. 보통 파이썬 N.M으로 번호가 붙은 파이썬 버전은 윈도우의 경우 C:\PyrkhonNM과 C:\PythonNM\lib\site-package에서, 유닉스와 리눅스의 경우는/usr/local/lib/pythonN.M/site-packeages와 /usr/local/lib/site-python에서 경로 파일을 주로 찾게 된다. 22장에서 경로 파일을 활용하여 sys.path 임포트 검색 경로를 설정하는 방법에 대해 더 많은 내용을 확인할 수 있다.

환경 설정은 선택 사항인데다 이 책이 운영체제 셸에 대한 책이 아니므로, 더 자세한 내용은 여기에서 다루지 않도록 하겠다. 더 자세한 내용은 여러분 시스템의 셸 매뉴얼이나 관련 문서를 참조하고 만약 환경 설정 과정에 문제에 봉착했다면, 시스템 관리자 또는 가까운 전문가에게 도움을 청하는 것이 좋다.

파이썬 명령 라인 인수

파이썬을 시스템 명령 라인(셸 프롬프트 또는 명령 프롬프트 창)으로 시작할 때, 파이썬이 코드를

어떻게 실행할 것인지에 대한 제어할 수 있도록 다양한 옵션 플래그를 전달할 수 있다. 이전에 다룬 내용이 시스템 전반에 영향을 미치는 환경 변수에 대한 것이라면, 명령 라인 인수는 스크립트를 실행시킬 때마다 달라질 수 있다. 3.3에서의 파이썬 명령 라인 호출의 전체 형태는 다음과 같다(2.7은 앞서 말한 몇 가지 차이점을 제외하고는 거의 유사하다).

```
python [-bBdEhiOqsSuvVWxX] [-c command | -m module-name | script | - ] [args]
```

이 절에서는 파이썬에서 가장 보편적으로 사용되는 인수들에 대하여 간단히 설명하려고 한다. 명령 라인 옵션에 대한 더 자세한 내용은 여기에서 다루지 않으므로 파이썬 매뉴얼 또는 참조 자료를 활용하기 바란다. 아니, 더 나은 방법은 파이썬 자체에 질문하는 것이다. 다음 명령어를 실행시켜보자.

```
C:\code> python -h
```

파이썬 도움말 화면이 뜨고 가능한 모든 명령어 옵션에 대한 설명을 확인할 수 있다. 복잡한 명령어를 다뤄야 한다면, 이 영역에서 표준 라이브러리 모듈인지에 대하여 반드시 확인하여야 한다. 보다 복잡한 명령어 처리 지원을 위해 기존 버전의 getop와 신생 argparse, 그리고 3.2 이후로는 사라진 optparse를 사용할 수 있다. 또한, 파이썬 라이브러리 매뉴얼과 다른 참조 자료에서 pdb와 profile 모듈에 대하여 확인할 수 있다.

인수를 사용하여 스크립트 파일 실행하기

대부분의 명령 라인은 바로 전 단락에서 보여 준 파이썬 명령 라인 형태에서 script와 args만을 사용하여 프로그램 소스 파일을 해당 프로그램에서 사용할 인수와 함께 실행한다. 설명을 위해 다음 스크립트(showargs.py라는 이름의 텍스트 파일로, C:\code 또는 여러분이 선택한 다른 디렉터리에 생성된다)를 생각해 보자. 이 스크립트는 파이썬 문자열의 리스트인 sys.argv처럼 스크립트에서 사용 가능한 명령 라인 인수를 출력한다(파이썬 스크립트 파일을 생성하거나 실행하는 방법을 아직 모른다면 2, 3장을 참조하면 된다. 여기에서 우리는 명령 라인 인수에 대해서만 다룰 것이다).

```
# showargs.py 파일
import sys
print(sys.args)
```

다음 명령 라인에서 python과 showargs.py는 절대 디렉터리 경로로 써줄 수도 있다. 전자는 PATH 의 경로에 있다고 가정하는 것이며, 후자는 현재 디렉터리에 스크립트 파일이 있음을 가정하고 있다. 스크립트를 위한 세 개의 인수(a b -c)가 sys.argv 리스트에 등장하고 여러분의 스크립트 코드에서 점검할 수 있다. sys.argv의 첫 번째 항목은 스크립트 파일명이 등장한다면, 항상 스크립트 파일명이다.

```
C:\code> python showargs.py a b -c        # 가장 일반적: 스크립트 파일 실행하기
['showargs.py', 'a', 'b', '-c']
```

이 책의 다른 부분에서 다루었듯이 파이썬의 **리스트**는 [대괄호]로, **문자열**은 '작은 따옴표'로 출력한다.

주어진 인수와 표준 입력으로 코드 실행하기

다른 코드 형태 사양 옵션으로는 파이썬 코드를 명령 라인 자체에서 실행하거나(-c), 표준 입력 스트림으로부터 실행시킬 코드를 받는 방법이 있다. - 기호는 파이프 또는 리다이렉트된 입력 스트림 파일에서 읽는 것을 의미한다. 이 용어들은 이 책의 다른 부분에서도 정의되어 있다.

```
C:\code> python -c "print (2 ** 100)"      # 명령어 인수로부터 코드를 읽어 들임
1267650600228229401496703205376

C:\code> python -c "import showargs"       # 이 코드를 실행하기 위해 파일을 임포트함
['-c']

C:\code> python - < showargs.py a b -c     # 표준 입력값으로부터 코드 읽어 들임
['-', 'a', 'b', '-c']

C:\code> python - a b -c < showargs.py     # 이전 라인과 동일한 결과를 출력함
['-', 'a', 'b', '-c']
```

검색 경로에서 모듈 실행하기

-m 옵션은 파이썬 모듈 검색 경로에서 모듈을 찾고, 최상위 레벨 스크립트로(모듈 __main__처럼) 실행시킨다. 즉, 이 옵션은 스크립트를 임포트할 때와 동일한 방식으로 검색한다. 일반적으로 sys.path로 알려진 디렉터리 리스트를 검색하게 되는데, 이 리스트에는 현재 디렉터리, PYTHONPATH에 설정된 디렉터리, 표준 라이브러리를 포함하고 있다. 여기에서는 '.py' 접미

사는 빼고 써야 하는데, 이는 파일명을 모듈로 취급하기 때문이다.

```
C:\code> python-m showargs a b -c          # 모듈을 스크립트처럼 찾고 실행시킴
['c:\\code\\showargs.py', 'a', 'b', '-c']
```

-m 옵션은 실행 도구들과, 상대적 임포트 구문의 사용 여부와 상관없이 패키지에 있는 모듈, .zip 압축 파일 내에 위치한 모듈을 모두 지원한다. 예를 들어, 이 옵션은 대화형 세션보다는 스크립트를 호출하는 명령 라인에서 pdb 디버거와 profile 프로파일러 모듈을 실행시킬 때 더 널리 사용되는 방법이다.

```
C:code> python                                          # 대화형 디버거 세션
>>> import pdb
>>> pdb.run('import showargs')
...뒷부분 생략: pdb 문서 참조

C:\code> python -m pdb showargs.py a b -c       # 스크립트 디버깅(c=continue)
> C:\code\showargs.py(2)<module>()
-> import sys
(Pdb) c
['showargs.py', 'a', 'b', '-c']
...뒷부분 생략. 종료하려면 q 입력
```

프로파일러는 여러분이 작성한 코드를 실행하고 실행 시간을 측정한다. 결과물은 파이썬, 운영체제, 컴퓨터 사양에 따라 다르게 나타날 수 있다.

```
C:\code> python -m profile showargs.py a b -c          # 스크립트 프로파일링하기
['showargs.py', 'a', 'b', '-c']
         9 function calls in 0.016 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
        2    0.000    0.000    0.000    0.000  :0(charmap_encode)
        1    0.000    0.000    0.000    0.000  :0(exec)
...뒷부분 생략. profile 문서 참조
```

-m 스위치를 이용해 표준 라이브러리에 위치한 IDLE GUI(3장에서 설명) 프로그램을 실행할 수 있다. 또한, 15장과 21장에서 했던 대로 명령줄에서 pydoc과 timeit 도구 모듈을 실행할 수도 있다(여기에서 실행한 도구들에 대한 자세한 설명은 각 장을 참조하면 된다).

```
c:\code> python -m idlelib.idle -n        # 패키지에서 IDLE 실행. 서브프로세스가 아님

c:\code> python -m pydoc -b               # pydoc과 timeit 도구 모듈 실행

c:\code> python -m timeit -n 1000 -r 3 -s "L=[1,2,3,4,5]" "M=[X+1 for x in L]"
```

최적화와 비버퍼 모드

파이썬은 'python' 바로 다음, 그리고 실행될 코드 지명 전에 추가 인수를 받아 자신의 행위를 제어한다. 이러한 변수들은 파이썬 자체가 받게 되는 것으로, 실행될 스크립트를 위한 것이 아니다. 예를 들어 -0은 파이썬을 최적화 모드에서 실행하게 하고, -u는 표준 스트림을 버퍼 없이 관리하도록 한다. 후자를 사용하면 어떤 출력된 텍스트라도 즉시 종료되고, 버퍼에서 딜레이되지 않는다.

```
C:\code> python -0 showargs.py a b -c    # 최적화: "pyo" 바이트 코드를 만들고 실행

C:\code> python -u showargs.py a b -c    # 비버퍼 방식의 표준 출력 스트림
```

실행 후 대화형 모드

마지막으로 -i 플래그는 스크립트 실행 후, 대화형 모드로 진입하게 된다. 특히 디버깅 도구에서 유용한데, 스크립트가 성공적으로 실행된 후 자세한 내용 확인을 위해 변수들의 최종 값을 출력할 수 있다.

```
C:\code> python -i showargs.py a b -c    # 스크립트 실행 후, 대화형 모드로 전환
['showargs.py', 'a', 'b', '-c']
>>> sys                                  # 임포트된 모듈인 sys의 최종 값
<module 'sys' (built-in)>
>>> ^z
```

또한 디버그 모드로 실행하지 않았더라도 스크립트가 예외가 발생하여 종료되었다면, 예외 발생 시점에 변수들은 어떤 값을 가지고 있는지 확인하기 위해 같은 방법으로 변수들을 출력할 수 있다. 물론, 디버거의 사후 도구를 여기에서 시작해도 된다(type은 윈도우에서 사용하는 파일 표시 명령어다. 다른 운영체제라면 cat이나 다른 명령어를 사용해야 한다).

```
C:\code> type divbad.py
X = 0
print(1 / X)

C:\code> python divbad.py                    # 버그 있는 스크립트를 실행
...오류 텍스트 생략
ZeroDivisionError: division by zero

C:\code> python -i divbad.py                 # 오류 시, 변숫값 출력
...오류 텍스트 생략
ZeroDivisionError: division by zero
>>> X
0
>>> import pdb                               # 전체 디버거 세션 시작
>>> pdb.pm()
> C:\code\divbad.py(2)<module>()
-> print(1 / X)
(Pdb) quit
```

파이썬 2.X 명령 라인 인수

이미 언급한 내용을 제외하고 파이썬 2.7은 3.X와의 호환성을 위해 추가 옵션을 지원하며(-3은 비호환성에 대해 경고하며, -Q는 나눗셈 연산자 모델을 제어한다), 일관되지 않은 들여쓰기된 부분에 대하여 탐지한다(3.X에서는 항상 감지/리포팅되는 부분이지만 2.X에서는 -t 옵션을 사용한다. 12장 참조). 이미 이야기했듯이, 이 주제에 대하여 더 궁금한 내용이 있다면 파이썬 2.X 자체에 질문할 수 있다.

```
C:\code> py -2 -h
```

파이썬 3.6 윈도우 런처 명령 라인

엄밀히 말하면, 이전 절에서는 파이썬 인터프리터 자체에 전달할 수 있는 인수에 대하여 설명하였다. 프로그램은 일반적으로 윈도우에서는 python.exe 이름을 가지고 있으며, 리눅스에서는 python이 그 이름이다(윈도우에서는 .exe가 일반적으로 생략된다). 다음 부록에서 보게 되겠지만, 윈도우 런처는 파이썬 3.6 그리고 이후 버전에 탑재되어 기능을 보강하거나 또는 독자적인 런처 패키지로 제공되기도 한다. 이는 파이썬과 스크립트를 시작하기 위해 사용되는 명령 라인에서 파이썬 버전 정보를 인수로 받아들이는 새로운 실행 프로그램을 추가한다(파일 what.py는 다음 부록에서 설명되어 있으며, 단지 파이썬 버전 번호를 출력한다).

```
C:\code> py what.py                          # 윈도우 런처 명령 라인
3.6.0

C:\code> py -2 what.py                       # 버전 번호 전환
2.7.3

C:\code> py -3.6 -i what.py -a -b -c         # py, python, 스크립트 모두를 위한 인수
3.6.0
>>> ^z
```

실제로, 앞 예제에서 마지막 실행에서와 같이 런처를 사용하는 명령 라인은 런처 자신이 사용할 인수(-3.6), 파이썬 자체가 사용할 인수(-i), 그리고 스크립트에서 사용할 인수(-a, -b, -c)를 모두 줄 수 있다. 런처는 또한 버전 번호를 스크립트 파일 가장 위에 #! 유닉스 라인에서부터 파싱할 수도 있다. 다음 부록에서 이 런처에 대하여 모두 다룰 예정이므로 나머지 내용은 다음 부록을 참조하는 것이 좋다.

더 많은 도움말

파이썬의 표준 매뉴얼 세트는 다양한 플랫폼에서 활용할 수 있는 팁을 제공한다. 표준 매뉴얼 세트는 윈도우 7 또는 이전 버전에서는 파이썬이 설치된 후 시작 버튼에서 확인할 수 있으며('Python Manuals'), 온라인으로는 https://www.python.org에서도 확인 가능하다. 매뉴얼 세트의 최상위 레벨의 절 중 'Using Python'을 보면 플랫폼에 특화된 도움말과 힌트를 확인할 수 있다. 여기에서 또한 최신의 플랫폼 간 연동 환경과 명령 라인의 세부 내용에 대하여서도 찾아볼 수 있다.

늘 그렇듯이 웹은 여러분의 조력자다. 특히 책이 업데이트되는 것보다 빠르게 진화하는 영역에 대해서는 더욱 그렇다. 파이썬이 널리 알려지거나 사용되고 있으므로 더 높은 수준의 활용 사례에 관련된 질문에 대한 대답을 웹에서 쉽게 찾을 수 있을 것이다.

APPENDIX

B

파이썬 3.6 윈도우 런처

이 부록에서는 파이썬의 새로운 윈도우 런처에 대하여 살펴보겠다. 파이썬 3.6에서는 자동으로 설치되지만, 이전 버전에서 사용할 수 있도록 웹에서 별도 패키지로 제공하고 있다. 비록 새로운 런처에는 잠재적 위험 요소가 있기도 하지만, 여러 파이썬이 공존하고 있는 컴퓨터에서 프로그램을 실행할 때 꼭 필요한 일관성을 제공한다.

이 페이지는 윈도우에서 파이썬을 사용하는 프로그래머를 위하여 작성하였다. 이는 근본적으로 플랫폼 특화된 영역이지만, 파이썬 입문자(이들 대부분은 윈도우에서 시작한다)와 윈도우와 유닉스 간 어디에서든 정상적으로 작동하는 코드를 작성하는 파이썬 개발자 모두를 대상으로 하는 도구다. 앞으로 살펴보겠지만, 새로운 런처는 윈도우에서 파이썬을 사용하거나 앞으로 사용하게 될 **모든 사람들**에게 영향을 끼칠 만큼 윈도우에서의 규칙을 근본적으로 바꾸었다.

유닉스가 남긴 유산

런처의 프로토콜을 제대로 이해하기 위해서는 짧은 역사 이야기로부터 시작해야 한다. 오래전 유닉스 개발자는 스크립트 코드 실행을 위한 프로그램을 지정할 수 있는 프로토콜을 고안했다. 유닉스 시스템에서 (리눅스와 맥 OS X을 포함하여) 스크립트 텍스트 파일의 첫 번째 줄이 #!로 시작한다면 이것은 특별한 의미를 가진다(종종 셔뱅(shebang)이라 불리는데, 바보 같은 표현이니 이 이름은 다시 언급하지 않겠다).

3장에서 이 주제에 대하여 간단히 다루었으나, 여기에서는 다른 관점에서 살펴보겠다. 유닉스 스크립트에서는 이 줄은 #! 다음에 코딩함으로써 나머지 스크립트의 내용을 실행할 프로그램을 지정한다. - #! 다음에 원하는 프로그램의 디렉터리 경로를 작성하거나, PATH 설정에 따라 대상을 검색하는 유닉스 유틸리티인 env를 호출하여 쓸 수 있다(PATH는 수정 가능한 시스템 환경 변수로 실행 파일을 검색할 디렉터리 리스트를 가지고 있다).

```
#!/usr/local/bin/python
...스크립트 코드 생략                          # 이 정해진 프로그램에서 실행

#!/usr/bin/env python
...스크립트 코드 생략                          # PATH에서 찾은 "python"으로 실행
```

이런 스크립트를 실행 가능하게 만들어(일례로, chmod +x script.py를 통해서) 명령 라인에서 단지 파일명만 입력하고도 실행시킬 수 있다. 가장 상위에 위치한 #!줄은 유닉스 셸에 파일의 나머지 코드를 실행시킬 프로그램을 알려 준다. 플랫폼의 설치 구조에 따라, #!줄에 명명된 python은 실제 실행 파일일 수도 있고, 다른 어딘가에 위치한 버전 특화된 실행 파일을 가리키는 심볼릭 링크일 수도 있다. 또한, 이 줄은 python3과 같이 특정 실행 파일을 명시적으로 명명할 수도 있다. 어느 방식이라도 #!줄, 심볼릭 링크 또는 PATH 설정을 변경함으로써, 유닉스 개발자는 스크립트를 적절한 파이썬으로 보낼 수 있다.

물론 이 중 어느 것도 윈도우 자체에는 적용되지 않는다. 윈도우에서는 #!줄이 아무 의미를 갖지 않는다. 파이썬 자체는 역사상 윈도우에서 이 줄이 등장하면 무시해 왔다(언어에서 '#'로 시작하면 주석을 의미한다). 여전히 파일별로 파이썬 실행 프로그램을 선택하는 것은 파이썬 2.X와 3.X가 함께 있는 컴퓨터에서는 강력한 특성이다. 많은 프로그래머들이 유닉스에로의 이식성을 위하여 #!줄을 코딩하는 것을 고려해 볼 때, 이 개념은 에뮬레이팅하기에 적합한 것으로 보인다.

윈도우가 남긴 유산

울타리 너머에서는 설치 모델이 매우 다르다. 과거에는(3.6 이전의 모든 파이썬에서는) 윈도우 인스톨러가 전역 윈도우 레지스트리를 변경하기 때문에 컴퓨터에 설치된 가장 최근 버전이 파이썬 파일을 클릭하거나, 명령 라인에서 직접 파일명을 실행할 때 파이썬 파일을 여는 버전이 된다.

일부 윈도우 사용자는 이 레지스트리를 제어판에서 기본 프로그램 설정으로 변경 가능한 파일명 연계로 알고 있기도 하다. 윈도우에서는 유닉스 스크립트에서처럼 실행 우선권을 가진 파일을 알려 줄 필요가 없다. 실제로, 윈도우에서는 그런 개념 자체가 존재하지 않으며, 파일명 연계와 명령어는 프로그램으로 파일을 실행하기에 충분하다.

이러한 설치 모델에서 만약 최근에 설치된 버전이 아닌 다른 버전으로 파일을 열고 싶다면, 원하는 파이썬의 전체 경로를 명령 라인에 주어서 실행하거나 파일명 연계를 원하는 버전으로 변경해 주어야 한다. 또한 PATH 설정을 통해 일반적인 python 명령 라인을 특정 파이썬으로 지정할 수 있으나, 파이썬이 이러한 설정을 하지 않았기 때문에 그 변경 내역은 아이콘 클릭이나 다른 방식으로 시작하는 스크립트에는 반영되지 않는다.

이는 윈도우에서의 자연적인 순서를 반영하고 있으며(.doc 파일을 클릭하면, 윈도우는 보통 설치된 최신 워드 프로그램에서 파일을 연다), 윈도우용 파이썬이 생긴 이래로 지금까지 이어 온 상태다. 하나의 머신에 다른 버전들이 요구되는 파이썬 스크립트를 가진다는 것은 별로 이상적이지 않지만, 최근에는 파이썬 2.X와 3.X 듀얼 체제에서 이러한 상황이 점점 보편화되어 가고 있으며 심지어 일반적인 일이 되고 있다. 윈도우에서 3.6 이전 버전으로 다중 파이썬을 실행하는 것은 개발자에게는 지루한 일이고, 입문자에게는 맥 빠지게 하는 일일 수 있다.

새로운 윈도우 런처 소개

파이썬 3.6(그리고 아마도 그 이후 버전)에 탑재되어 자동으로 설치되거나 또는 다른 버전에서 사용할 수 있도록 독자적인 패키지로도 출시된 새로운 윈도우 런처는 두 개의 새로운 실행 프로그램을 제공함으로써 이전 설치 모델의 부족한 부분을 해결한다.

- py.exe: 콘솔 프로그램용
- pyw.exe: 콘솔이 아닌 (일반적으로 GUI) 프로그램용

이 두 프로그램은 각각 .py와 .pyw 파일을 열기 위해 윈도우 확장자 연결을 통해 등록된다. 파이썬의 원래 메인 프로그램인 python.exe와 같이(이 두 프로그램들이 반대하지는 않지만, 대체로 포함하는) 이 두 실행 프로그램 또한 바이트 코드 파일을 바로 시작할 수 있도록 등록된다. 이들의 무기 중 몇 가지를 정리하면, 이 두 실행 프로그램은 다음과 같은 성질을 갖는다.

- 윈도우 파일명 연계를 통해 아이콘 클릭이나 파일명 명령어로 시작된 파이썬 소스나 바이트 코드 파일을 자동으로 열 수 있다.
- 일반적으로 시스템 검색 경로에 설치되어, 명령 라인에서 사용하기 위해 별도의 디렉터리 경로나 PATH 설정을 할 필요가 없다.
- 스크립트나 대화형 세션을 시작할 때, 파이썬 버전이 명령 라인 인수로 쉽게 전달되도록 한다.
- 파일의 코드를 어느 파이썬 버전으로 실행할 것인지를 결정하기 위해 스크립트의 최상위에 있는 유닉스 스타일의 #! 주석을 파싱하는 것을 시도한다.

그 결과 새로운 런처에서는 윈도우에 여러 버전의 파이썬이 설치되어 있을 때, 설치된 최신 버전에 제약을 받거나 명령 라인에 명시적으로 전체 경로로 특정 버전을 지정할 필요가 없다. 대신 파일별 또는 명령어별로 명시적으로 버전을 선택할 수 있으며, 두 경우에 부분적으로 또는 전체 형태로 버전을 지정할 수 있다. 이는 다음과 같은 방식으로 동작한다.

1. **파일별** 버전을 선택하려면 유닉스 스타일로 스크립트 상단에 다음과 같은 주석을 사용한다.

```
#! python2
#! /usr/bin/python2.7
#! /usr/bin/env python3
```

2. **명령어별** 버전을 선택하려면 다음과 같은 형태의 명령 라인을 사용한다.

```
py -2 m.py
py -2.7 m.py
py -3 m.py
```

예를 들어, 이 기법 중 **첫 번째**는 스크립트가 의존하고 있는 파이썬 버전을 선언하기 위한 일종의 지시자로서의 역할을 할 수 있다. 이는 스크립트를 명령 라인 또는 아이콘 클릭 어느 것으로 실행하든 런처에 의해 적용된다(이는 script.py 파일의 변형이다).

```
#!python3
...
...a 3.X 스크립트                    # 3.X의 설치 버전 중 최신 버전으로 실행
...

#!python2
```

```
...
...a 2.X 스크립트                    # 2.X의 설치 버전 중 최신 버전으로 실행
...

#!python2.6
...
...a 2.6 스크립트                    # 오직 2.6 버전으로만 실행
...
```

윈도우에서 명령 라인은 명령 프롬프트 창에서 타이핑되며, 이 부록에서 C:\code> 프롬프트에 의해 지정된다. 다음 중 첫 번째 아이템은 파일명 연계로 인해 두 번째 아이템과 아이콘 클릭과 동일하다.

```
C:\code> script.py                   # 파일당 #! 라인이 있다면 해당 버전으로, 아니면 기본 버전으로 실행
C:\code> py script.py                # 상동. 하지만 py.exe가 명시적으로 실행됨
```

또는 두 번째 기법으로 명령 라인에서 인수를 변경하여 버전을 선택할 수 있다.

```
C:\code> py -3 script.py             # 최신 3.X 버전으로 실행
C:\code> py -2 script.py             # 최신 2.X 버전으로 실행
C:\code> py -2.6 script.py           # 2.6 버전에서만 실행
```

이는 스크립트를 시작하거나 대화형 인터프리터(어떤 스크립트도 명명되지 않았을 때)를 시작할 때 적용된다.

```
C:\code> py -3                       # 최신 3.X 버전으로 시작. 대화형
C:\code> py -2                       # 최신 2.X 버전으로 시작. 대화형
C:\code> py -3.1                     # 3.1 버전으로 시작. 대화형
C:\code> py                          # 기본 파이썬으로 시작(처음에 2.X: 다음 내용 참조)
```

만약 파일 내에 #! 라인이 있는데 이 파일을 시작하는 명령 라인에 다른 버전 번호가 있다면, 명령 라인의 버전 번호가 파일 내의 지시자보다 우선한다.

```
#! python3.2
...
...a 2.X 스크립트
...

C\code> py script.py                 # 파일 내 명령어대로 3.2에서 실행
C\code> py -3.1 script.py            # 3.2가 있더라도 3.1에서 실행
```

런처는 또한 파이썬 버전이 누락되거나 일부만 기술되어 있을 때, 특정 파이썬 버전을 선택하기 위해 **발견법**(heuristics)을 적용한다. 예를 들어, 버전이 2로만 특정지어지면 최신의 2.X 버전이 실행되고, #! 라인에서 버전을 명명하지 않은 파일을 아이콘 클릭이나 일반적인 명칭의 명령 라인(예 py m.py, m.py)으로 시작하면 2.X로 실행이 된다. 이 대신 3.X로 실행이 되게 하려면, PY_PYTHON이나 파일 엔트리 설정을 변경해야 한다(더 자세한 내용은 다음에 나올 내용 참조).

특히 현재의 2.X/3.X 듀얼 체제에서 명시적인 버전 선택은, 대부분의 입문자가 언어를 처음으로 접하게 되는 윈도우를 위한 유용한 추가 사항으로 볼 수 있다. 물론 여기에도 잠재적 위험 요소가 존재하는데, 인식할 수 없는 유닉스 #! 라인이나 기본 실행 버전인 2.X를 혼동하는 등의 문제가 이에 해당한다. 하지만 이는 한 머신에서 2.X와 3.X가 우아하게 공존할 수 있도록 하며, 명령 라인에서 버전을 합리적으로 제어할 수 있도록 해준다.

윈도우 런처에 대한 전체 이야기는 (여기에서는 압축 또는 생략한 고급 특성이나 활용 예시를 포함하여) 파이썬의 릴리즈 노트나 PEP(제안 문서)를 찾기 위해 웹 검색으로 확인하면 된다. 여러 가지 중에 몇 가지 특성을 들자면, 런처에서는 32와 64비트 설치를 선택할 수 있고 설정 파일의 기본값을 명시하며, 사용자 정의의 #! 명령어 문자열 확장을 정의한다.

윈도우 런처 사용 지침서

유닉스 스크립트 작성이 익숙한 일부 독자들은 이전 절의 내용으로도 시작하기에 충분할 것이다. 나머지 독자들을 위해 이번 절에서는 사용 지침서의 형태 추가적인 내용을 실제 작동하는 런처의 구체적인 예제를 통해 더듬어 나갈 수 있도록 구성하여 제공할 것이다. 또한, 이 절에서는 부가적인 런처의 세부 사항에 대해서 설명하고 있다. 때문에 유닉스에 익숙해진 베테랑들도 자신들이 작성한 파이썬 스크립트를 윈도우 머신으로 복사해 넣기 전에 잠깐만 살펴보면 큰 도움을 얻게 것이다.

그 시작으로 다음의 간단한 스크립트 what.py를 사용해 보자. 이 스크립트는 이 코드를 실행할 파이썬 버전 넘버에 따라 2.X와 3.X에서 모두 동작 가능하다. 이 코드에서는 sys.version을 이용하는데, 이를 공백 분리한 후 처음 등장하는 문자열이 파이썬의 버전 넘버다.

```
#!python3
import sys
print(sys.version.split()[0])            # 문자열의 첫 부분
```

이 글의 내용을 따라가며 진행하려면 해당 스크립트 코드를 텍스트 파일 편집기에 입력하고, 실행할 명령 라인을 입력할 명령 프롬프트 창을 열고 스크립트를 저장한 디렉터리로 이동하면 된다(C:\code는 내가 작업하는 디렉터리지만, 여러분이 원하는 어느 곳에나 저장해도 된다. 윈도우 사용 팁에 대한 자세한 내용은 3장을 참조하자).

이 스크립트 첫 번째 줄의 코멘트는 필요한 파이썬 버전을 지정한다. 이 줄은 유닉스 관례에 따라 반드시 #!로 시작해야 하며, python3 전에 공백을 두어도 되고 두지 않아도 무방하다. 내 컴퓨터에는 현재 파이썬 2.7, 3.1, 3.2, 3.6이 설치되어 있다. 다음 절에서 파일 내 지시자, 명령 라인, 그리고 기본값에 대하여 순서대로 살펴보면서 이 스크립트의 첫 번째 줄이 수정됨에 따라 어느 버전이 호출되는지를 확인해 보자.

1단계: 파일 내 버전 지시자 사용하기

이 스크립트가 코딩된 대로 아이콘 클릭 또는 명령 라인으로 실행될 때, 첫 번째 줄은 등록된 py.exe 런처에게 설치된 버전 중 최신의 3.X 버전을 사용하여 실행할 것을 지시한다.

```
#! python3
import sys
print(sys.version.split()[0])

C:\code> what.py                          # 파일 지시자대로 실행
3.6.0

C:\code> py what.py                       # 상동. 최신의 3.X로 실행
3.6.0
```

다시 말하지만, #! 뒤에 공백을 넣는 것은 선택 사항이다. 이에 대하여 다시 설명하기 위해 이 예제에서는 띄어 썼다. 처음 what.py는 아이콘 클릭이나 전체 명령어인 py what.py와 동일하다는 점에 주목하자. 이는 런처가 설치될 때 윈도우 파일명 연계 레지스트리에 .py 파일을 자동으로 여는 프로그램으로 py.exe가 등록되어 있기 때문이다.

또한, 런처 관련 문서(이 부록을 포함하여)에서 **최신** 버전이라 함은 **가장 높은 번호**의 버전을 의미함을 기억하자. 이는 가장 최근에 컴퓨터에다 설치된 버전이 아니라, 가장 최근에 출시된 최신 버전을 말한다(예를 들자면 여러분이 3.6을 설치 후 3.1을 설치하였다더라도 #!python3은 전자를 선택한다). 런처는 여러분 컴퓨터에 설치된 파이썬을 돌아다니며, 여러분이 기술한 버전 또는 기본 버전에 적합한 가장 높은 숫자의 버전을 찾는다. 이것은 이전에 다룬 최종 설치 버전이 선택되는

모델과 다르다.

이제, 첫 번째 줄의 이름을 python2로 바꾸면, 설치 버전 중 최신의(가장 높은 번호의) 2.X 버전이 실행된다. 다음의 내용으로 확인하자. 우리의 스크립트 중 마지막 두 줄은 변경되지 않으므로 생략한다.

```
#!python2
...스크립트의 나머지 부분은 변경되지 않음

C:\code> python what.py        # #!에 의해 최신 2.X 로 실행
2.7.3
```

필요하다면 특정 버전을 요구할 수도 있는데, 예를 들어 파이썬 버전 중 최신 버전을 원하지 않는 경우에 사용된다.

```
#!python3.1
...

C:\code> python what.py        # #!에 의해 3.1로 실행
3.1.4
```

이것은 요구한 버전이 설치되어 있지 않더라도 유효하다. 이는 런처에 의해 오류로 처리된다.

```
#! python2.7
...

C:\code> python what.py
Requested python version (2.7) is not installed
```

인식되지 않는 유닉스 #!줄은 이를 보완하기 위해 명령 라인 스위치로 버전 넘버를 제공하지 않는 한 에러로 취급된다. 다음 절에서 이에 대하여 더 상세하게 알아보겠다.

```
#!/bin/python
...

C:\code> what.py
Unable to create process using '/bin/python "C:\code\what.py" '

C:\code> py what.py
Unable to create process using '/bin/python what.py'
```

```
C:\code> py -3 what.py
3.6.0
```

기술적으로, 런처는 스크립트 파일의 최상단에 위치한 유닉스 스타일의 #!줄을 다음 네 개의 패턴 중 하나에 따라 인식한다.

```
#!/usr/bin/env python*
#!/usr/bin/python*
#!/usr/local/bin/python*
#!python*
```

앞의 인식 가능하고 해석 가능한 형태 중 하나에도 해당하지 않는 #!줄이라면, 파일 실행 프로세스를 시작하는 명령 라인에서 버전을 지정하여 윈도우에 전달할 것이라고 가정한다. 만약 이것이 윈도우 명령어에서 유효하지 않다면 앞에서 본 에러 메시지를 생성한다(또한 런처는 설정 파일을 통해 사용자 정의 명령어로의 확장을 지원하는데, 인식 불가능한 명령어를 윈도우에 전달하기 전에 이 사용자 정의 명령어를 시도한다. 여기서는 이에 대해서 그냥 넘어가겠다).

인식 가능한 #!줄에서 디렉터리 경로는 유닉스 관례에 따라 작성함으로써 유닉스 플랫폼으로의 이식성을 보장한다. 앞서 나열한 인식 가능한 패턴의 마지막 * 부분에 파이썬 버전을 아래의 세 가지 형태 중 하나로 표기하면 된다.

부분 표기(예 python3)

주어진 메이저 릴리즈 번호를 가진 버전 중, 가장 높은 번호를 가진 마이너 릴리즈 번호의 버전을 실행할 때 사용한다.

전체 표기(예 python3.1)

지정된 버전으로만 실행한다. 선택적으로 -32라는 접미어를 덧붙이면 32-비트 버전을 우선한다는 것을 의미한다(예 python3.1-32).

생략(예 python)

런처의 기본 버전으로 실행한다. PY_PYTHON 환경 변수를 3으로 설정하여 변경하지 않는다면 기본 버전은 2다. 앞서 이야기한 또 다른 위험 요소다.

#!줄이 없는 파일은 통칭의 python과 동일하게 동작하며, (앞서 언급한 버전 생략의 경우와 같이) PY_PYTHON의 기본 설정값에 따라 실행 버전이 선택된다. 첫 번째 부분 표기의 경우에는 버전 특화된 환경 설정값에 영향을 받는다(예를 들어 PY_PYTHON3를 3.1로 설정하면 python3 실

행 시 3.1이 선택되며, PY_PYTHON2를 2.6으로 설정하면 python2를 실행하면 2.6이 선택된다). 기본 버전에 대해서는 이 사용 지침서에서 나중에 다시 다룰 것이다.

첫째, #!줄 포맷의 * 부분 다음에 오는 것은 모두 파이썬 자신(즉, 프로그램 python.exe)에 전달되는 명령 라인 인수로 가정한다. 만약 py 명령 라인에서 #!줄의 인수를 대체할 것으로 간주되는 인수를 주지 않는다면 말이다.

```
#!python3 [python.exe 관련 인수는 모두 여기에 위치함]
...
```

여기에는 부록 A에서 보았던 모든 파이썬 명령 라인 인수를 포함한다. 이는 우리를 일반적으로 런처 명령 라인으로 이끄며, 다음 절로 자연스럽게 넘어가기 충분하다.

2단계: 명령 라인 버전 스위치를 사용하기

앞서 언급했듯이 명령 라인에서의 버전 스위치는 파일에 파이썬 버전이 표기되어 있지 않은 경우, 파이썬 버전을 선택하기 위해 사용할 수 있다. 레지스트리에 파일명 연계나 파일 내의 #!줄에 버전 정보를 주지 않고도, py와 pyw 명령 라인을 실행하여 버전을 스위치에 전달할 수 있다. 다음에서는 스크립트를 수정하여 #! 지시자를 삭제하였다.

```
# 런처 지시자 없음
...

C:\code> py -3 what.py                    # 명령 라인 스위치에 따라 실행
3.6.0

C:\code> py -2 what.py                    # 상동. 설치 버전 중 최신 2.X
2.7.3

C:\code> py -3.2 what.py                  # 상동. 3.2 버전으로만 실행
3.2.3

C:\code> py what.py                       # 런처의 기본값으로 실행(앞의 내용 참조)
2.7.3
```

또한, 명령 라인 스위치는 파일 내 지시자에 의한 버전 지정에 우선한다.

```
#! python3.1
...

C:\code> what.py                          # 파일 지시자에 따라 실행
3.1.4

C:\code> py what.py                       # 상동
3.1.4

C:\code> py -3.2 what.py                  # 스위치가 지시자에 우선함
3.2.3

C:\code> py -2 what.py                    # 상동
2.7.3
```

형식상 런처는 다음 명령 라인 인수 유형을 받는다(이는 정확히 앞선 절에서 설명한 파일의 #!줄의 마지막 * 부분과 일치한다).

```
-2              최신 파이썬 2.X 버전 시작
-3              최신 파이썬 3.X 버전 시작
-X.Y            특정 버전 시작(X는 2 또는 3)
-X.Y-32         특정 버전의 32비트 파이썬으로 시작
```

그리고 런처의 명령 라인은 다음의 일반적인 형태를 가지고 있다.

```
py [py.exe arg] [python.exe args] script.py [script.py args]
```

런처 자신의 인수 다음에 나오는 것이라면 모두 python.exe에 전달되는 것으로 다루면 된다. 이는 전형적으로 파이썬 자신에 대한 인수를 포함하고 있으며, 스크립트 파일명과 스크립트를 위한 인수들 앞에 위치한다.

보통 –m mod, -c cmd, - 프로그램 지정 형태는 부록 A에서 다룬 다른 파이썬 명령 라인 인수들이 모두 그러하듯이 py 명령 라인에서도 동작한다. 앞서 언급한 대로 python.exe에 대한 인수는 파일에서 #! 지시자 맨 끝에 등장할 수 있다. 다만 py 명령 라인에 등장하는 인수가 #! 지시자보다 우선한다.

어떻게 동작하는지 알아보기 위해 이전 명령 라인 인수를 확장하여 새로운 스크립트를 작성해 보자. sys.argv가 이 스크립트 고유의 인수이며, 파이썬(python.exe)에 스크립트가 실행된 후 대화형 프롬프트(>>>)로 이동하도록 –i 스위치를 사용한다.

```
# args.py와 사용자가 입력한 인수도 함께 표시
import sys
print(sys.version.split()[0])
print(sys.argv)

C:\code> py -3 -i args.py -a 1 -b -c          # -3: py, -i: 파이썬, 나머지: 스크립트용
3.6.0
['args.py', '-a', '1', '-b', '-c']
>>> ^Z

C:\code> py -i args.py -a 1 -b -c             # 파이썬과 스크립트용 인수
2.7.3
['args.py', '-a', '1', '-b', '-c']
>>> ^Z

C:\code> py -3 -c print(99)                    # -3: py, 나머지: 파이썬: "-c cmd"
99

C:\code> py -2 -c "print 99"
99
```

처음 두 경우에서 스크립트 자체에 #!줄이 없는 상태에서 명령 라인에서 버전이 주어지지 않는다면 기본 파이썬이 어떻게 실행되는지 주목하자. 우연히도, 이 내용은 이 지침서의 마지막 주제로 우리를 인도한다.

3단계: 기본값 변경하기

이미 언급했듯이 런처의 기본값은 2.X로 #! 지시자에 특정 버전 번호가 없다면 통칭의 python 명령어로 2.X가 실행된다. 이는 유닉스의 전체 경로(예 #!/usr/bin/python)로 등장하든 아니든 (#!python) 그렇다. 후자의 경우 실제 동작하는 것을 확인할 수 있게 원래의 what.py 스크립트를 다음과 같이 코딩하였다.

```
#!python
...                                  # #!/usr/bin/python과 동일

C:\code> what.py                     # 런처 기본값에 의해 실행
2.7.3
```

기본값은 지시자가 없는 경우에도 적용된다. 아마도 대부분의 경우는 주로 윈도우에서 또는 윈도우에서만 사용되도록 작성한 코드일 것이다.

```
# 런처 지시자가 없음
...

C:\code> what.py              # 역시 기본값에 따라 실행됨
2.7.3

C:\code> py what.py           # 상동
2.7.3
```

여기에서 초기화 파일 또는 환경 변수 설정을 통해 런처의 기본값을 3.X로 바꿀 수 있다. 이는 윈도우 레지스트리에서 py.exe 또는 pyw.exe와의 이름 연계를 통해 명령 라인이나 아이콘 클릭으로 파일을 실행할 때 모두 적용된다.

```
# 런처 지시자가 없음
...

C:\code> what.py              # 기본값에 따라 실행
2.7.3

C:\code> set PY_PYTHON=3      # 또는 제어판 => 시스템을 통해서 설정 가능
C:\code> what.py              # 변경된 기본값에 따라 실행
3.6.0
```

앞서 제안했듯이 보다 상세한 제어를 위해 버전 특화된 환경 변수를 설정할 수 있는데, 부분적으로 지정된 버전을 설치된 릴리즈 중 가장 높은 마이너 번호를 갖는 릴리즈를 선택하는 대신 특정 릴리즈를 가리키도록 할 수 있다.

```
#!python3
...

C:\code> py what.py                     # 3.X 중 '최신' 버전 실행
3.6.0

C:\code> set PY_PYTHON3=3.1             # 2.X를 위해서는 PY_PYTHON2를 사용할 것
C:\code> py what.py                     # 가장 높은 마이너 릴리즈보다 우선함
3.1.4
```

set은 명령어 프롬프트 창에서만 적용되는 것으로, 컴퓨터 전체에서 적용이 되려면 제어판의 시스템 창에서 설정하여야 한다(이 설정에 대한 도움말은 부록 A를 참조할 것). 실행할 대부분의 파이썬 코드가 무엇이냐에 따라 이러한 방식으로 기본값 설정하는 것을 원할 수도, 원하지 않을 수도 있다. 많은 파이썬 2.X 사용자는 아마도 기본값을 변경하지 않고 사용하면서 필요 시 #! 줄이나 py 명령 라인을 이용해 실행 프로그램 버전을 변경할 수 있다.

하지만 지시자가 없는 파일을 위해 사용되는 설정값인 PY_PYTHON은 매우 중요하다. 대부분 윈도우에서 파이썬을 사용해 온 프로그래머들은 아마도 3.6을 설치한 후 기본값으로 3.X를 기대할 것이다. 특히, 처음으로 3.6에 의해 설치된 런처를 고려해 본다면 말이다. 겉보기에는 역설적인 이 내용에 대해서는 다음 절에서 알아보자.

새로운 윈도우 런처의 함정

3.6의 새로운 윈도우 런처를 추가한 것은 멋진 일이지만, 3.X에서의 대부분의 내용과 마찬가지로 수 년 전에 등장했다면 더 좋았을 것이다. 불행히도, 이는 몇 가지 하위 버전과는 호환되지 않는데, 이는 여러 버전이 공존하는 파이썬 세계에서는 불가피한 부작용으로 기존 프로그램 중 일부는 이로 인해 제대로 동작하지 않을 수 있다. 이러한 부작용은 내가 예전에 쓴 책이나 다른 책들에서 등장하는 예제에도 포함되어 있을 것이다. 코드를 3.6으로 포팅하는 동안 다음 세 가지 이슈와 맞닥뜨렸다.

- 인식되지 않는 유닉스 #!줄은 윈도우에서 스크립트 실행이 되지 않는다.
- 런처의 기본값은 별도로 이야기하지 않으면 2.X를 사용한다.
- 새로운 PATH 확장은 기본값에 의해 무효화되어 모순으로 여겨진다.

이 절의 나머지에서는 이 세 가지 이슈 각각에 대하여 차례대로 설명하겠다. 다음에서 《프로그래밍 파이썬》 네 번째 개정판의 프로그램을 예제로 런처의 비호환성이 미치는 영향에 대하여 설명하는데, 이들 3.1과 3.2 예제를 3.6으로 포팅하는 과정이 신규 런처를 처음 경험하는 것이었기 때문이다. 이 경우, 3.6을 설치하는 것은 3.2와 3.1 버전에서 작성된 수많은 예제 프로그램을 망치게 되는 것이다. 여기에 요약된 이러한 실패의 원인은 여러분이 작성한 코드에도 동일한 영향을 줄 것이다.

위험 요소 1: 인식되지 않은 #!줄

새로운 윈도우 런처는 #!/usr/bin/env python으로 시작하는 유닉스 #!줄을 인식하는데 다른 일반적인 유닉스 형태인 #!/bin/env python(실제 일부 유닉스에서는 강제사항이기도 하다)을 인식하지는 않는다. 과거에는 후자를 사용하는 스크립트도 윈도우에서 내 책의 몇몇 예제들을 포함해 동작하였다. 왜냐하면 유닉스 호환성을 위해 코딩된 #!줄은 지금까지 모든 윈도우용 파

이썬에서는 주석으로 인식되어 무시되었기 때문이다. 이 스크립트들은 이제 3.6에서 실행되지 않는데, 새로운 런처가 이들 지시자 형태를 인식하지 못하여 에러 메시지를 내기 때문이다.

보다 일반적으로, 인식되지 않는 #! 유닉스 줄을 포함한 스크립트라면 이제는 윈도우에서 실행되지 않는다. 여기에는 첫 번째 줄이 앞서 설명한 네 개의 인식 가능한 패턴 중 하나를 따르지 않는 #!줄을 갖는 모든 스크립트가 포함된다. /usr/bin/env python*과 /usr/bin/python*, /usr/local/bin/python*, python* 이외의 다른 것은 모두 동작하지 않으며, 코드 변경이 필요하다. 예를 들어 다소 일반적인 #!/bin/python 줄 또한 윈도우에서 실행이 실패하게 되며, 이를 방지하기 위해 명령 라인 스위치를 통해 버전 번호를 제공해야 한다.

유닉스 스타일의 #!줄은 아마 윈도우 전용 프로그램에는 존재하지 않을 것이다. 하지만 유닉스에서도 동작하도록 만들어진 프로그램이라면 꽤 일반적인 코딩 방식이다. 이러한 인식되지 않는 유닉스 지시자를 윈도우에서 에러로 간주하는 것은 다소 과하다 여겨질 수 있다. 특히, 3.6에서 새롭게 추가된 행위라는 점을 고려한다면 기대 못한 일일 수 있다. 왜 지금까지 모든 윈도우 파이썬이 그래왔듯이 그냥 인식되지 않는 #!줄을 무시하고 기본 파이썬으로 파일을 실행하지 않는 걸까? 아마도 미래의 3.X 버전에서는 이 점이 개선이 될 수도 있겠다(이 문제에 대하여 다소 연기가 될 수도 있다). 하지만 지금으로서는 윈도우에서 파이썬 3.6과 함께 설치된 런처를 기반으로 한 실행 환경에서 #!/bin/env나 다른 인식할 수 없는 패턴을 사용하는 모든 파일은 변경되어야만 한다.

책 예제의 영향도와 해결 방안

내가 3.6으로 포팅한 책의 예제들의 경우, 이 문제는 대략 십여 개의 #!/bin/env python로 시작하는 스크립트에서 발생하였다. 유감스럽게도 여기에는 이 책의 사용자 친화적인 최상위 레벨의 데모용 런처 스크립트(PyGadget이나 PyDemo)가 포함되어 있다. 이를 해결하기 위해 이 문장을 #!/usr/bin/env python 포맷으로 변경하였다. 윈도우의 파일 연결에서 런처를 생략하는 것도 또 하나의 방법이 될 수 있다(예 .py 파일을 py.exe 대신에 python.exe로 연결해 두는 것). 하지만 이 방식은 런처가 가지는 장점을 이용할 수 없는 방식으로, 특히 입문자들에게는 다소 많은 것을 요구하는 방식으로 보인다.

여기서 한 가지 문제가 있다. 이상하게도 어떤 명령줄 스위치를 런처에 전달해도(python.exe 인수인 경우에도) 이러한 효과가 나타나지 않으며, 기본 파이썬 프로그램이 실행된다. m.py와 py m.py 둘 다 인식되지 않는 #!줄에서 에러를 발생시키지만, py –i m.py는 기본 파이썬으로 파

일을 실행한다. 이는 런처의 버그로 보일 수 있지만, 기본 런처로 해결이 가능하다. 다음 주제에서 더 알아보자.

위험 요소 2: 런처는 기본적으로 2.X를 사용

이상하게도, 윈도우 3.6 런처는 3.X를 명시적으로 선택하지 않은 스크립트를 실행할 때 기본값으로 설치된 파이썬 2.X 버전을 사용한다. 즉 #! 지시자가 없거나, python 이름을 사용한 스크립트가 아이콘 클릭에 의해 실행되거나, 명령줄에서 파일 이름을 직접 지정(m.py)하거나, 런처 명령줄에 버전 스위치를 지정하지 않은 경우(py m.py)에는 일반적으로 2.X 파이썬이 실행된다는 것이다. 이는 여러분 컴퓨터에 3.6이 2.X보다 나중에 설치되었다 하더라도 동일하며, 이로 인해 초기에는 많은 3.X용 제대로 실행되지 않을 수 있다.

이 영향은 잠재적으로 광범위하다. 예를 들면 지시자가 없는 3.X 파일을 3.6 설치 후에 아이콘을 클릭하여 실행하면 실행되지 않는데, 이는 연결된 런처가 이를 기본값인 2.X로 실행하라는 의미로 받아들이기 때문이다. 파이썬을 처음 다루는 일부 사용자들에게는 이것이 그리 유쾌한 일은 아닐 것이다. 이는 3.X 파일이 명시적으로 python3이라는 버전 넘버를 제공하는 #! 지시자가 없는 것으로 가정하지만, 대부분의 윈도우에서의 실행을 목적으로 작성된 스크립트는 #!줄을 가지고 있지 않으며, 런처가 출시되기 전에 작성된 수많은 파일들은 자신이 기대하는 버전 번호를 포함하고 있지 않을 것이기 때문이다. 따라서 대부분의 3.X 사용자는 기본적으로 3.6 설치 후, PY_PYTHON을 설정해야 한다. 이는 매우 불편하다.

명시적으로 버전이 지정되지 않은 프로그램을 실행하는 것은 유닉스에서도 모호하긴 마찬가지여서 이 경우 python과 특정 버전과의 심볼릭 링크에 의존하게 된다(오늘날 대부분은 2.X로 연결된다. 마치 새로운 윈도우 런처의 상태가 따라가는 것처럼 보인다). 하지만 이전 이슈에 대해서라면, 이전에는 정상적으로 동작하던 스크립트가 3.6에서 새로운 오류를 발생시키지는 않을 것이다. 대부분의 프로그래머는 유닉스 주석이 윈도우에서 영향을 미칠 것이라 생각하지는 않을 것이다. 또한, 3.X가 설치되었음에도 기본으로 2.X가 사용될 것이라고도 예상하지 못할 것이다.

책 예제의 영향도와 해결 방안

예제 포팅 관점에서 2.X를 기본 프로그램으로 한다는 것은 3.6 설치 후 3.6 스크립트에서 여러 가지 실패의 요인이 된다. #!줄이 없는 스크립트나, 유닉스 호환 가능한 #!/usr/bin/python 줄을 가지고 있는 두 스크립트 모두에서 말이다. 후자 스크립트에서의 문제를 해결하기 위해 이

카테고리의 모든 스크립트에서 python을 python3로 명시적으로 바꿔주어야 한다. 전자와 후자의 스크립트를 모두 한 번에 해결하려면, 윈도우 런처의 기본값을 3.X로 변경해 주어야 한다. py.ini 설정 파일(자세한 내용은 런처 관련 문서를 참조할 것)을 변경하거나 또는 PY_PYTHON 환경 변수를 이전 예제에서와 같이 설정하여(C:\> set PY_PYTHON = 3) 변경할 수 있다. 앞에서 언급했듯이 파일 연계를 수작업을 바꾸어 주는 것도 하나의 해결책이 될 수 있지만, 이들 중 어느 하나도 이전 설치 방식으로 인한 방법보다 단순해 보이진 않는다.

위험 요소 3: 새로운 PATH 확장 방안

새로운 런처를 설치한 것 외에, 윈도우 파이썬 3.6 설치 프로그램은 자동적으로 3.6의 python.exe 실행 파일을 포함한 디렉터리를 시스템 PATH 변수에 추가한다. 이는 윈도우 입문자들의 편의를 위한 것으로 추론해 볼 수 있다. 이로써 이들은 python의 전체 디렉터리 경로 대신 python이라고만 타이핑하면 된다. 이는 시스템 공학 측면에서 갖는 런처의 특성이 아니며, 따라서 일반적으로 이 점이 스크립트의 실패 요인이 될 수 없다. 이 특징은 책의 예제에 어떠한 영향도 주지 않는다. 하지만 단지 python이라 입력하는 것은 런처의 동작 및 목표와 충돌할 수 있으므로 가능하면 피하는 것이 좋다. 다소 미묘한 특징이라 그 이유에 대하여 좀 더 설명하도록 하겠다.

앞에서도 설명했지만 새로운 런처의 py와 pyw 실행 프로그램은 기본적으로 시스템 검색 경로에 설치되고, 이를 실행하기 위해 별도의 디렉터리 경로나 PATH 설정이 필요하지는 않다. 만약 스크립트를 python 대신에 py 명령 라인으로 시작한다면, 새로운 PATH 특성은 신경 쓰지 않아도 된다. 실제로, py는 대부분의 콘텍스트상에서 python을 완전히 **포괄한다**. 파일 연계가 python 대신 py나 pyw를 실행시킨다는 점을 고려해 볼 때, 여러분도 아마 똑같이 하려 할 것이다. py 대신에 python을 사용하는 것은 중 불필요한 중복성과 불일치성만 확인하게 될 것이며, 더구나 런처 콘텍스트상에서 사용된 버전과 다른 버전을 시작할 수도 있다. 이로써 이 두 방식 간의 설정이 점점 더 싱크가 맞지 않게 된다. 요약하면 python을 PATH에 추가하는 것은 새로운 런처의 관점에서는 **모순되는 특징**으로, 잠재적으로 봤을 때 에러의 요인으로 작용할 수 있다.

PATH를 업데이트하는 것은 여러분이 python 명령어가 일반적으로 3.6을 실행하기 바란다는 것을 가정하는데, 이 특징은 기본적으로 **활성화되어 있지 않다**는 점에 주목할 필요가 있다. 만약 PATH 업데이트를 원한다면, 설치 화면에서 이를 잊지 말고 선택하여야 한다(만약 원하지 않

으면 아무것도 건드리지 않으면 된다!). 두 번째 위험 요소 때문에 많은 사용자들은 PY_PYTHON을 3으로 설정하여 런처 아이콘을 실행했을 때 실행할 파이썬 버전을 설정해야 한다. 이것은 런처의 사용 목적 중 하나인 PATH를 설정하는 것보다 간단하지 않다. 가장 나은 방법은 런처의 실행 파일을 그냥 사용하면서 필요 시 PY_PYTHON을 변경하는 것이다.

결론: 윈도우를 위한 이점

공정하게 말하면 이전 절에서 다룬 위험 요소 중 일부는 윈도우에서 유닉스 특성과 다중 설치 버전을 동시에 지원하기 위한 노력에서 나온 불가피한 결과일지도 모른다. 그 대신에 여러 버전이 혼재되어 있는 스크립트와 설치 작업 관리를 위한 일관된 방법을 제공한다. 3.6이나 그 이후 버전에 탑재된 윈도우 런처를 일단 사용하기 시작했다면 주요 자산임을 알게 될 것이며, 처음에 맞닥뜨릴 호환성 관련 문제를 무사히 넘길 수 있게 된다.

실제로, 윈도우 스크립트에도 유닉스 호환성을 위하여 #! 라인에 명시적으로 버전 번호를 넣어 코딩하는 습관을 들이기를 원할 것이다(예 #!/usr/bin/python3). 이는 윈도우에서 적절히 실행되기 위해 여러분의 코드에서 필요로 하는 요구 사항을 선언할 뿐 아니라, 런처의 기본값을 파괴함으로써 여러분이 작성한 스크립트가 이후에 유닉스에서도 실행 가능하도록 만들 것이다.

하지만 런처는 일부 이전에는 #!줄을 가졌음에도 유효했던 스크립트를 실행 불가하게 만들고, 여러분이 기대하지 않았거나 또는 여러분이 작성한 스크립트가 사용할 수 없는 기본 버전을 선택하기도 한다. 이런 위험 요소를 딕셔너리에서 제거하고 싶다면 시스템 환경 설정과 코드 변경이 수반되어야 함을 인지해야 한다. 새로운 상사가 옛 상사보다 낫지만, 결국은 같은 부류인 것과 마찬가지다.

윈도우에서의 사용법에 대하여 더 알고 싶다면 부록 A의 설치와 시스템 설정 방법에 대한 부분을 참조하자. 또한 3장에서 일반적인 개념을 확인할 수 있으며, 파이썬 매뉴얼에서도 플랫폼에 특화된 문서를 찾아보면 된다.

APPENDIX

C

파이썬 변경 사항 참조

이 부록에서는 최신 파이썬 버전에서의 변경 사항에 대하여 해당 변경 사항이 처음 등장한 개정판을 기준으로 간단하게 정리 요약하였으며, 이 책 어디에서 다루고 있는지에 대한 정보를 함께 제공한다. 이는 이전 개정판을 읽은 독자와 함께 이전 파이썬 버전을 전환 개발해야 하는 개발자 모두에게 유익한 참고 자료가 될 것이다.

여기서 파이썬의 변화가 이 책의 최근 개정판들과 어떻게 연관되는지를 보여 준다.

- 2013년 **다섯 번째** 개정판은 파이썬 3.3과 2.7을 다룸
- 2009년 **네 번째** 개정판은 파이썬 2.6과 3.0(일부 3.1의 특성도 함께)을 다룸
- 2007년 **세 번째** 개정판은 파이썬 2.5를 다룸
- 1999년의 **초판**과 2003년 **두 번째** 개정판은 각각 파이썬 2.0과 2.2를 다룸
- 이 책의 조상 격인, 1996년 《**프로그래밍 파이썬**》은 파이썬 1.3을 다룸

따라서 다섯 번째 개정판에서의 변경 사항에만 관심이 있다면, 앞으로 다룰 내용 중 파이썬 2.7, 3.2 그리고 3.6에서의 변경 사항만 확인하면 된다. 네, 다섯 번째 개정판과 관련한(즉 세 번째 개정판 이후에 일어난) 변경 사항은 파이썬 2.6, 3.0, 3.1의 변경 사항을 함께 확인하면 된다. 세 번째 개정판에서의 변경 사항은 아주 간단하게 언급했는데, 이는 현재로서는 역사적 가치만을 지녔기 때문이다.

더불어 이 부록은 주요 변경 사항과 책에 미치는 영향도에 초점을 맞추고 있으며, 파이썬의 진화 과정에 대한 전체 안내서가 아님을 미리 밝힌다. 각 파이썬 릴리즈별 더 자세한 변경 이력이 궁금하다면 이들의 표준 문서 중 'What's New' 문서를 참조하면 되며, python.org의 Documents 메뉴에서도 확인할 수 있다. 참고로 이 책 15장에서 파이썬 관련 문서와 표준 매뉴얼에 대하여 다루고 있다.

2.X/3.X의 주요 차이점

이 부록의 대부분은 파이썬의 변경 사항과 이 책에서 다루는 내용을 관련짓는다. 만약 그 대신에 2.X와 3.X의 가장 중요한 차이점에 대하여 간략히 요약된 내용을 찾는다면, 다음 내용으로 충분할 것이다. 이 절은 주로 3.X와 2.X 의 가장 최근 릴리즈인 3.6과 2.7을 비교하였다. 많은 3.X의 특성은 여기에 언급되지 않는데, 이들은 이미 2.6에도 추가되었거나(C1 with문, 클래스 데코레이터) 또는 이후 2.7에도 함께 포팅되었기(C1 집합과 딕셔너리 컴프리헨션) 때문이다. 하지만 그 전의 2.X 릴리즈에서는 사용 가능하지는 않다. 이보다 이전 버전에서의 변경 사항에 대한 상세 내역은 이 후의 절을 참조하고, 미래의 파이썬에서 등장하게 될 변경 사항은 파이썬의 'What's New' 문서를 참조하기 바란다.

3.X 차이점

파이썬 버전 전체에서 달라진 도구들에 대하여 요약해 보면 다음과 같다.

- **유니코드 문자열 모델**: 3.X에서 일반적인 str 문자열은 모든 유니코드 텍스트를 지원하며, 여기에는 아스키와 8비트의 바이트 시퀀스에 해당하는 분절된 bytes 타입이 포함된다. 2.X에서 일반적인 str 문자열은 아스키를 포함한 8비트 텍스트와 선택 사항인 유니코드의 리치 텍스트에 해당하는 분절된 unicode 형태 둘 다를 지원한다.
- **파일 모델**: 3.X에서 open에 의해 생성된 파일은 내용에 따라 특화된다. 텍스트 파일은 유니코드 인코딩을 구현하고 내용은 str 문자열로 표현하며, 바이너리 파일은 bytes 문자열로 그 내용을 나타낸다. 2.X에서 파일은 별도의 인터페이스를 사용한다. open으로 생성된 파일은 8비트 텍스트 또는 바이트 기반의 데이터이든지 그 내용을 str 문자열로 나타내지만, codecs.open은 유니코드 텍스트 인코딩을 구현한다.

- **클래스 모델**: 3.X에서 모든 클래스는 자동으로 object로부터 파생하여 수많은 변경 사항과 새 형식 클래스의 확장이 가능해졌다. 여기에는 이전과는 다른 상속 알고리즘, 내장된 실행, 다이아몬드 패턴 트리를 위한 MRO 검색 순서도 포함된다. 2.X에서 일반적인 클래스는 **전형적인** 모델을 따르지만, 선택적으로 object 또는 다른 내장된 타입으로부터의 명시적인 상속으로 새로운 형식의 모델도 가능하다.
- **내장된 반복 객체**: 3.X에서 map, zip, range, filter 그리고 딕셔너리 keys, values, items는 모두 반복 객체로 요청에 따라 값을 생성한다. 2.X는 이 호출들이 물리적 리스트를 만든다.
- **프린팅**: 3.X는 키워드 인수를 갖는 내장된 함수로 설정하지만, 2.X는 특정 문법의 문장으로 설정한다.
- **상대적 임포트**: 2.X, 3.X 모두 상대적 임포트 문장인 from을 지원하지만, 3.X는 검색 규칙을 변경하여 일반적인 임포트를 수행할 때 패키지가 위치한 디렉터리를 건너뛰도록 하였다.
- **진짜 나누기(True division)**: 2.X와 3.X는 // 반내림 연산자를 지원한다. 하지만 /의 경우 3.X는 분수 나머지를 포함한 진짜 나누기인 반면, 2.X에서는 타입에 따라 다르다.
- **정수 타입**: 3.X는 확장된 정확도를 지원하는 싱글 정수 타입을 가지고 있다. 2.X는 일반적인 int와 확장된 long, 그리고 long으로의 자동 전환을 지원한다.
- **복합 타입 범위**: 3.X에서는 모든 포괄 타입(리스트, 집합, 딕셔너리, 제너레이터)은 변수를 해당 표현식 안으로 지역화한다. 2.X에서 list 포괄 타입은 변수를 지역화하지 않는다.
- **PyDoc**: 모든 브라우저에서 적용 가능한 pydoc –b 인터페이스는 3.2에서는 지원 사항이지만 3.6에서는 필수 사항이다. 2.X에서는 이 대신에 원래의 pydoc –g GUI 클라이언트 인터페이스를 사용한다.
- **바이트 코드 저장소**: 3.2 기준 현재, 3.X는 바이트 코드 파일을 버전 식별 이름으로 소스 디렉터리의 하위 디렉터리인 __pycache__에 저장한다. 2.X에서 바이트 코드는 일반 명칭으로 소스 파일 디렉터리에 저장된다.
- **내장된 시스템 예외**: 3.6 기준 현재, 3.X에서는 OS와 IO 클래스를 위해 예외 계층을 재작업하여 카테고리와 세부 항목을 추가하는 작업을 진행하였다. 2.X에서 예외 속성은 때로는 시스템 오류에서 검사되어야만 한다.
- **비교와 정렬**: 3.X에서는 서로 다른 데이터 타입 간 또는 딕셔너리 간 상대적 크기 비교를 하는 것은 오류이며, 정렬은 데이터 타입이 섞여 있는 경우나 일반적인 비교 함수를 지원하지 않는다(그대신 key 매퍼를 사용한다). 2.X에서는 이 모든 형태가 정상적으로 동작한다.

- **문자열 예외와 모듈 함수**: 문자열 기반의 예외는 3.X에서는 모두 삭제되었으며, 2.X에서도 2.6부터는 사라지고 대신에 클래스를 사용한다. string 모듈 함수는 문자열 객체 메서드와 중복되어 3.X에서는 삭제되었다.
- **언어 삭제**: 표 C-2에 따라 3.X는 수많은 2.X의 언어 요소를 삭제하고, 재명명하거나 재배치하였다(reload, apply, `x`, < >, 0177, 999L, dict.has_key, raw_input, xrange, file, reduce, file.xreadlines).

3.X에서만 볼 수 있는 확장

3.X에서만 가능한 도구들에 대하여 요약해 보면 다음과 같다.

- **확장된 시퀀스 할당**: 3.X는 시퀀스 할당 대상에 *를 허용함으로써 리스트에서 매치되지 않고 남은 반복 객체를 모을 수 있다. 2.X는 슬라이싱으로 유사한 효과를 낼 수 있다.
- **nonlocal**: 3.X는 nonlocal문을 제공하여 유효 함수 범위에 있는 이름을 내포 함수 안에서 변경할 수 있도록 한다. 2.X는 이와 비슷한 작업을 하기 위해 함수 속성, 가변 객체 그리고 클래스 상태 정보를 활용한다.
- **함수 주석**: 3.X에서는 함수 인수와 반환값의 데이터 타입을 함수에서 유지는 되지만 사용되지는 않는 객체를 이용하여 주석 처리를 할 수 있다. 2.X는 별도 객체 또는 데코레이터 인수를 활용하여 유사한 작업을 할 수 있다.
- **키워드 전용 인수**: 3.X는 키워드로 전달되어야만 하는 함수 인수에 대한 명세를 허용한다. 이는 주로 부가적인 설정 옵션에 사용된다. 2.X에서는 이를 위해 인수 분석과 딕셔너리 팝을 사용한다.
- **Exception chaining**: 3.X에서는 예외들이 연쇄적으로 발생하는 것과 raise from 확장을 이용하여 에러 메시지에 등장하는 것을 허용한다. 3.6은 이 연쇄 반응을 취소할 수 있게 None을 허용한다.
- **Yield from**: 3.6 현재 yield문은 from을 사용하여 내포된 제너레이터에 권한을 위임할 수 있다. 2.X의 경우, 간단한 활용 예제에서는 for 반복문으로 동일한 효과를 얻을 수 있다.
- **Namespace 패키지**: 3.6부터는 폴백 옵션으로 패키지 모델이 패키지가 초기화 파일 없이 다중 디렉터리에 걸쳐서 존재하는 것을 허용하도록 확장되었다. 2.X에서는 임포트를 확장하여 유사하게 만들 수 있다.

- **윈도우 런처**: 3.6에서는 런처가 윈도우용 파이썬에 탑재되었으며, 2.X를 포함하여 이전 버전의 파이썬에서도 사용할 수 있도록 별도로 설치 가능하다.
- **내부 작업**: 3.2 부터 쓰레딩은 가상 머신 명령어 카운트 대신, 타임 슬라이스로 구현된다. 그리고 3.6에서는 유니코드 텍스트를 고정 길이 바이트 대신에 변동 길이 방식으로 저장한다. 2.X의 문자열 모델은 일반적으로 유니코드 사용을 최소화한다.

3.X 변경 사항에 대한 일반적인 설명

이 책의 최근 개정판에서 다루었던 파이썬 3.X 버전은 큰 틀에서 이전 2.X 버전과 동일한 언어라 볼 수 있지만, 몇 가지 중요한 부분에서 차이를 보인다. 서문에서 논의하고 이전 절에서도 요약했듯이, 3.X의 필수적인 유니코드 모델, 새 형식 클래스, 제너레이터에 대한 광범위한 강조, 다른 기능적 도구들 만으로도 3.X는 실질적으로 다른 경험이 될 것이다.

전반적으로 파이썬 3.X는 더 깔끔한 언어이지만, 여러 가지 면에서 볼 때, 상당히 진보된 개념을 기반으로 하고 있어 더 **복잡한 언어**이기도 하다. 실제로 이러한 변경 사항 중 일부는 여러분이 파이썬을 배우기 위해 이미 파이썬을 알고 있을 것이라 가정한다. 서문에서 3.X에서 중요한 돌고 도는 지식의 의존성에 대하여 언급하였으며, 이는 앞으로 나올 주제들의 의존성을 암시하기도 한다.

예로, 3.X에서 list 호출로 딕셔너리 뷰를 감싸야 하는 근거는 매우 미묘하며, 최소한 뷰, 제너레이터, 반복 프로토콜에 대해 상당한 선견지명이 요구된다. 이와 유사하게 키워드 인수는 단순한 도구(예 프린팅, 문자열 포맷, 딕셔너리 생성, 정렬)에서 요구되기 때문에 입문자 입장에서는 함수에 대해 전부 이해할 정도로 배우기 전에 등장하게 된다. 이 책의 목표 중 하나는 2.X와 3.X 듀얼 버전 체계에서 이러한 지식의 격차에 다리 역할을 하는 것이다.

라이브러리와 도구에서의 변경 사항

파이썬 3.X에는 이 책에 영향을 주지 않는다는 이유로 여기 부록에서는 다루지 않은 추가적인 변경 사항들이 있다. 예를 들어, 일부 표준 라이브러리와 개발 도구는 이 책의 핵심 언어 범위 밖에 있어, 일부는 내용 전개상 언급되기도 하였지만(예 timeit), 그 외의 내용은 항상 여기에서 다루고 있다(예 PyDoc).

완성도를 위해 다음 절부터 이 범주들에서 3.X의 성장에 대하여 언급하였다. 이 범주들의 일부 변경 내역은 처음 소개된 파이썬 버전과 책 판본과 함께 이 부록의 후반부에서도 열거하였다.

표준 라이브러리 변경 사항

공식적으로는 파이썬 표준 라이브러리는 파이썬에서 늘 사용 가능하며, 실제 파이썬 프로그램에 퍼져 있음에도 이 책의 핵심 언어 주제는 아니다. 실제로, 라이브러리는 3.2 개발 기간 동안 벌어진 한시적 3.X 언어 변경 유예에 영향을 받지 않았다.

이 때문에 표준 라이브러리의 변경 사항은 활용에 초점을 맞춘 《프로그래밍 파이썬》 같은 책에 더 많은 영향을 준다. 비록 대부분의 표준 라이브러리 기능은 현재에도 존재하지만, 파이썬 3.X는 모듈의 이름을 바꾸거나, 모듈을 묶어서 패키지로 만들거나, API 호출 패턴을 변경하는 등 무분별하게 고쳤다.

일부 라이브러리 변경 내역은 더 광범위하다. 일례로 파이썬 3.X의 유니코드 모델은 3.X 표준 라이브러리에서 광범위한 차이점을 만들어내는데, 이는 잠재적으로 파일 내용, 파일명, 디렉터리 탐색, 파이프, 디스크립터 파일, 소켓, GUI에서의 텍스트, FTP/email 같은 인터넷 프로토콜, CGI 스크립트, 많은 종류의 웹 콘텐츠, 심지어 DBM 파일, shelve, pickle과 같은 도구들을 처리하는 모든 프로그램에 영향을 미친다.

3.X 표준 라이브러리의 변경 내역에 대하여 종합적으로 확인하려면, 파이썬 매뉴얼 중에 3.X 릴리즈의(특히 3.0) 'What's New' 문서를 참조하기 바란다. 《프로그래밍 파이썬》은 도처에서 파이썬 3.X를 사용하고 있어, 3.X 라이브러리 변경 내역에 대한 가이드 역할을 할 수 있다.

도구 변경 사항

대부분의 개발 도구들은 2.X와 3.X 간에 동일하지만(예 디버깅, 프로파일링, 실행 시간 측정하기, 시험하기), 몇몇은 3.X에서 언어와 라이브러리에 따라 변화를 겪었다. 이들 중, PyDoc 모듈의 문서 시스템은 3.2와 이전 버전의 GUI 클라이언트 모델에서 옮겨져서 웹 브라우저 인터페이스로 대체되었다.

다른 주목할 만한 변경 사항으로는 제3자 소프트웨어 설치 및 배포에 활용되는 disutils 패키지가 3.X에서 새로운 패키징 시스템에 포함되었고, 이 책에서 설명한 새로운 __pycache__ 바이트 코드 저장소 방식은 개선이 되었음에도 잠재적으로 많은 파이썬 도구와 프로그램들에 영향을 주고 있으며, 스레딩의 내부 구현 방식은 3.2 기준으로 전역 인터프리터 락(GIL)을

가상 머신의 명령어 카운터 대신에 절대 타임 슬라이스를 사용하도록 수정하여 논쟁의 여지를 줄였다.

3.X로의 전환 개발

만약 파이썬 2.X에서 파이썬 3.X로 전환 구축을 하고 있다면, 파이썬 3.X에 탑재되어 있는 2to3 자동 코드 전환 스크립트에 대해 반드시 확인해 보자. 파이썬 설치 폴더 내에 Tools\Scripts나 웹 검색을 통해서 확인이 가능하다. 이 스크립트가 모든 것을 전환해 주지는 않으며, 주로 핵심 언어 코드를 전환해 준다. 3.X 표준 라이브러리 API는 좀 더 다를 것이다. 여전히 이 스크립트는 대부분의 2.X 코드를 3.X에서 실행할 수 있도록 합리적으로 전환한다.

역으로, 파이썬 3.X 코드를 2.X에서 실행할 수 있도록 바꿔 주는 3to2 프로그램을 서드파티에서 찾아볼 수 있다. 개발 목적과 제약 조건으로 인해 양쪽 파이썬 버전 모두에서 동작할 수 있도록 코드를 유지, 관리해야 한다면 2to3와 3to2 둘 중 어느 것이라도 유용할 것이다. 보다 상세한 내역과 추가적인 도구와 기법에 대해서는 웹을 통해 확인하도록 하자.

또한, 이 책에서 설명한 기법을 이용하여 2.X와 3.X 양쪽에서 자유롭게 실행 가능하게 코드를 짤 수도 있다. __future__로부터 3.X의 특성을 임포트하거나, 버전 특화된 도구의 사용을 기피하는 등의 방법을 적용할 수 있다. 이 책의 많은 예제들은 플랫폼에 영향을 받지 않도록 작성되었다. 예를 들어, 21장 벤치마킹 도구, 25장 모듈 리로더와 콤마 포매터, 31장 클래스 트리 리스터, 38장과 39장에서 보여 준 대부분의 데코레이터 예제, 41장 말미의 농담 스크립트 등등이 있다. 2.X와 3.X의 핵심 언어의 차이점을 이해하고 있는 한, 때로는 이 예제에 맞춰 코딩하는 것이 간단하다.

만약 2.X와 3.X에서 모두 동작하도록 코드를 작성하는 일에 관심이 있다면, 도구들을 버전 간 매핑하고 명칭을 변경해 주는 라이브러리인 six를 참조하는 것이 좋다. 이는 http://packages.python.org/six에서 다운받을 수 있다. 이 패키지는 언어의 의미론과 라이브러리 API에서의 모든 차이를 상쇄할 수는 없으며, 많은 경우 직접 파이썬이 버전 이식성을 실현하는 대신에 이 패키지의 라이브러리 도구를 활용해야 한다. 그 대신에 이 라이브러리 도구를 사용하면 버전 중립적인 프로그램을 작성할 수 있게 된다.

다섯 번째 개정판에서의 파이썬 변경 사항: 2.7, 3.2, 3.3

다음에 기술된 변경 사항은 네 번째 개정판이 출간된 후에 파이썬 2.X와 3.X 버전에서 발생하여 이번 개정판에 편입된 변경 내역이다. 특히, 이 절은 파이썬 2.7, 3.2, 3.3에서 파이썬 책과 관련한 변경 사항에 대하여 기술하였다.

파이썬 2.7에서의 변경 내역

기술적인 면에서 파이썬 2.7은 이 책의 이전 개정판에서는 3.X만의 특징으로 다루었던 것 중 일부를 역으로 포팅하여 거의 편입시켰다. 다섯 번째 개정판에서 이들은 2.7의 도구로 설명할 것이다. 이 중에는 다음과 같은 것들이 포함된다.

- 집합 리터럴:

```
{1, 4, 2, 3, 4}
```

- 컴프리헨션 집합과 딕셔너리:

```
{c * 4 for c in 'spam'}, {c: c * 4 for c in 'spam'}
```

- 선택적 메서드로 편입된 딕셔너리 뷰:

```
dict.viewkeys(), dict.viewvalues(), dict.viewitems()
```

- str.format에서의 콤마 구분자와 필드 자동 번호 할당(3.1부터):

```
'{:,.2f} {}'.format(1234567.891, 'spam')
```

- 중첩 with문 콘텍스트 매니저(3.1부터):

```
with X() as x, Y() as y: ...
```

- repr의 출력 내용 개선(3.1부터 역으로 포팅됨: 뒤의 내용 참조)

이 주제들이 이 책 어디에서 다루어졌는지 확인하려면, 앞으로 나올 표 C-1의 3.0 변경 내역 리스트의 아이템들이나 파이썬 3.1 변경 사항 절에서 찾아보면 된다. 이들은 이미 3.X에 존재하지만, 2.7에서도 사용할 수 있도록 업데이트되었다.

논리적인 측면에서 현재 계획에 따르면 2.7은 2.X 버전의 마지막 메이저 버전이 될 예정이나,

실제 제품화 작업에서 지속적으로 사용되고 있어 그 운영 기간은 길 것이다. 2.7 이후에 새로운 개발물은 파이썬 3.X 버전으로 이관될 예정이다.

그렇기는 하지만 2.X가 여전히 넓은 사용자 지지 기반을 가지고 있음을 고려하면, 이 공식적 입장이 오랜 세월을 지나도 건재할 것인지에 대하여 예견하는 것은 불가능하다. 이에 대해서는 서문에서 더 알아보도록 하자. 예를 들어, 최적화된 PyPy는 여전히 2.X로만 구현되어 있다. 또는 몬티 파이썬 라인 'I'm not dead yet...'을 차용하자면, 파이썬 2.X 이야기의 전개에서 지속적으로 관심을 가지고 있어야 한다.

파이썬 3.3에서의 변경 내역

파이썬 3.3은 마이너 버전 업그레이드로 보기에는 놀랄 만큼 많은 변경 내역을 포함하고 있다. 이 중 일부는 3.X의 이전 버전에서 작성한 코드와 완전히 호환되지 않는다. 이들 중 3.3의 필수 요소로 설치된 윈도우 런처는 윈도우에서 실행되는 기존 3.X 스크립트를 망가뜨릴 가능성이 있다.

여기에 간략하게 3.3의 변경 내역 중 주목할 만한 내용에 대하여 이 책에서 다룬 위치와 함께 설명하도록 하겠다. 파이썬 3.3의 변경 내역은 다음과 같다.

- 2.X에서는 많이 차지하던 **메모리 공간**을 새로운 가변 길이의 문자열 저장 방식과 속성 명칭 공유용 딕셔너리 시스템으로 절감하였다(37장과 32장 참조).
- 새로운 **namespace 패키지 모델**을 도입해 새로운 스타일의 패키지가 여러 디렉터리에 걸쳐 저장됨으로써 __init__.py 파일이 필요하지 않게 되었다(24장 참조).
- 서브제너레이터에 위임하기 위한 새로운 구문이 도입되었다(yield from ...(20장)).
- 예외 콘텍스트 억제를 위해 새로운 구문이 도입되었다(raise ... from None(34장)).
- 전환 구축을 쉽게 하기 위해 2.X의 유니코드 리터럴 형태를 받아들일 수 있는 새로운 구문을 도입하였다. 3.3은 이제 2.X의 유니코드 리터럴 u'xxxx'를 일반 문자열인 'xxxx'와 동일하게 취급한다. 이는 2.X와 3.X의 바이트 리터럴 b'xxxx'를 일반 문자열인 'xxxx'와 동일하게 취급하는 것과 유사하다(4장, 7장, 37장 참조).
- OS와 IO의 **예외 계층** 구조를 재작업하여, 보다 포괄적이고 일반적인 슈퍼클래스와 예외 객체 속성을 검사할 필요가 없는 일반 오류를 위한 새로운 서브클래스를 제공할 수 있게 구성하였다.

- pydoc –b로 시작되는 PyDoc 문서에 대한 모든 웹 브라우저 기반의 인터페이스가 윈도우 7 또는 이전 버전의 스타트 버튼 또는 pydoc –g로 시작되는 독립형 GUI 클라이언트 검색 인터페이스를 대체하였다(15장).
- 오랫동안 이어 온 표준 라이브러리 모듈 중 일부가 변경되었다. ftplib, time, email, 그리고 아마도 distutils가 여기에 포함되며, disutils는 이 책에도 영향을 미친다. time의 경우, 3.X에서는 플랫폼 간 이식성을 고려한 새로운 호출을 제공한다(21, 39장).
- importlib.__import__의 __import__ 함수를 구현하여 부분적으로는 그 구현물을 통합하고 더 분명히 드러나게 한다(22장, 25장).
- 윈도우 3.3 인스톨러는 설치 시 옵션으로 시스템 PATH 설정에 3.3 디렉터리를 포함하도록 확장하여, 일부 명령 라인을 단순화한다(부록 A, B).
- 새로운 윈도우 런처로 기능이 강화되었다. 윈도우 런처는 윈도우에서 파이썬 스크립트를 작업 할당할 때 유닉스 스타일의 #!줄을 해석하고, #!줄과 새로운 py 명령 라인으로 2.X와 3.X 버전 중 특정 버전을 파일별 또는 명령어별로 명시적으로 선택할 수 있게 한다(부록 B).

파이썬 3.2에서의 변경 내역

파이썬 3.2는 3.X 버전의 진화를 계속한다. 3.2는 3.X의 핵심 언어의 변경 활동을 중지한 기간 동안에 개발되어 3.2 관련 변경 사항은 작은 편이다.

- 바이트 코드 파일 저장소 모델 변경: __pycache__(2장, 22장)
- struct 모듈의 문자열 자동 인코딩 기능 제거(9장, 37장)
- 3.X str/bytes를 분리하여 파이썬 자체의 지원을 더 잘 받게 됨(이 책에서는 다루지 않음)
- cgi.escape 호출은 3.2+로 옮겨짐(이 책에서는 다루지 않음)
- 스레딩(Threading) 구현 방식 변경: 타임 슬라이스(이 책에서는 다루지 않음)

네 번째 개정판의 변경 내역: 2.6, 3.0, 3.1

네 번째 개정판은 파이썬 3.0과 2.6에 대하여 다루고 있으며, 3.1에서 이루어진 약간의 주요 변경 사항에 대하여 포함하고 있다. 3.0과 3.1의 변경 내역은 다섯 번째 개정판의 파이썬 3.3을 포함하여 3.X 버전의 향후 릴리즈에도 적용될 것이며, 2.6의 변경 내역 또한 2.7의 일부분을

차지한다. 앞에서도 이야기하였지만, 여기 3.X의 변경 사항 중 일부는 향후에 파이썬 2.7로 역으로 포팅된다(예 집합 리터럴, 집합 컴프리헨션 및 딕셔너리).

파이썬 3.1에서의 변경 내역

다음 절에서는 3.0과 2.6의 변경 사항에 더하여, 네 번째 개정판 출판을 앞두고 곧 출시될 파이썬 3.1의 주요 확장 내용에 대하여 보강하였다.

- 문자열 format 메서드 호출에서의 콤마 분리자와 자동 필드 번호 할당(7장)
- with문에서의 다중 콘텍스트 매니저 구문(34장)
- 숫자 객체를 위한 새로운 메서드(5장)
- (다섯 번째 개정판 전에는 언급되지 않았음) 부동 소수점 표현 방식이 변경됨(4장, 5장)

이 다섯 번째 개정판은 이들 주제에 대하여 괄호 안의 장에서 다루고 있다. 파이썬 3.1이 주로 최적화를 목적으로 하고 있어 3.0 출시 후 상대적으로 빠른 시일 내에 출시되어, 네 번째 개정판 또한 바로 3.1을 적용하였다. 실제로 파이썬 3.1은 3.0을 모두 대체하고 가장 최신의 파이썬이 일반적으로 받아서 사용하기 가장 좋기 때문에 그 개정판에서 사용한 '파이썬 3.0'이란 단어는 일반적으로 파이썬 3.0에서 도입되었으나, 이번 판본의 3.3을 포함하여 3.X 버전 전체에 존재하는 언어의 변형이라 이해하면 된다.

한 가지 예외를 짚고 넘어가자면 네 번째 개정판에서는 3.1의 새로운 특성인 **부동 소수점** 숫자를 위한 repr 표현 방식에 대하여 다루지 **않았다**. 새로운 표현 알고리즘은 부동 소숫점 숫자를 가능하면 보다 적은 십진수(하지만 경우에 따라서는 더 많아지기도 함)로 더 똑똑하게 표현하려고 시도한다. 이 사항에 대해서는 이번 다섯 번째 개정판에 반영하였다.

파이썬 3.0과 2.6의 변경 내역

네 번째 개정판의 언어 변화는 파이썬 3.0과 2.6으로부터 기인한다. 2.6의 모든 변경 사항과 3.0의 변경 내역은 오늘날의 파이썬 2.7과 3.3에도 동일하게 적용된다. 파이썬 2.7은 2.6에는 없었던 3.0의 일부 특성들이 추가, 보강되었으며(이 부록 앞의 내용 참조) 파이썬 3.3은 3.0에서 도입된 모든 특성을 상속한다.

초기 3.X 버전에는 많은 변경이 있었기 때문에 다음 표로 이 책에서 상세 내역이 나와 있는 위치와 함께 간단히 정리하였다. 표 C-1은 3.X 변경 사항 중 첫 번째 부분으로, 네 번째 개정판에서 다룬 가장 중요한 신규 특징에 대하여 열거하였다. 이와 함께 현재의 다섯 번째 개정판에서 이에 대하여 설명하고 있는 주요 장을 함께 기술하였으니, 세부 내역은 이를 참고하기 바란다.

표 C-1 파이썬 2.6과 3.0의 확장 내역

확장 내역	관련 챕터
3.0에서의 print 함수	11
3.0에서의 nonlocal x,y문	17
2.6, 3.0에서의 str.format 메서드	7
3.0에서의 문자열 타입: 유니코드 텍스트용 str, 바이너리 데이터용 bytes	7, 37
3.0에서의 텍스트, 바이너리 파일 구분	9, 37
2.6과 3.0에서의 클래스 데코레이터: @private('age')	32, 39
3.0에서의 새로운 반복자: range, map, zip	14, 20
3.0에서의 딕셔너리 뷰: D.key, D.values, D.items	8, 14
3.0에서의 나눗셈 연산자: remainders, /, //	5
3.0에서의 집합 리터럴: {a, b, c}	5
3.0에서의 집합 컴프리헨션: {x**2 for x in seq}	4, 5, 14, 20
2.6, 3.0에서의 2진수 문자열 지원: 0b0101, bin(l)	5
2.6, 3.0에서의 분수 타입: Fraction(1, 3)	5
3.0에서의 함수 주석달기: def f(a:99, b:str)->int	19
3.0에서의 키워드 전용 인수: f(a, *b, c, **d)	18, 20
3.0에서의 확장된 시퀀스 분석: a, *b = seq	11, 13
3.0에서의 상대 경로를 이용한 패키지 임포트 구문: from .	24
2.6, 3.0에서의 콘텍스트 매니저: with/as	34, 36
3.0에서의 예외 구문 변경: raise, except/as, superclass	34, 35
3.0에서의 예외 연쇄 처리: raise e2 from e1	34
2.6과 3.0에서의 예약어 변경	11
3.0에서의 새 형식 클래스 변경	32
2.6과 3.0에서의 프로퍼티 데코레이터: @property	38
2.6과 3.0에서의 디스크립터 사용	32, 38
2.6과 3.0에서의 메타클래스 사용	32, 40
2.6과 3.0에서의 추상화 기반 클래스 지원	29

3.0에서의 특정 언어 삭제

3.X에서는 설계 정리의 일환으로 2.X에서 지원되던 얼마간의 언어 도구들이 삭제되었다. 표 C-2는 3.X에서 사라진 항목 중 이 책에 영향을 주는 항목에 대하여 이 개정판에서 관련 있는 장과 함께 정리하였다. 표에서 볼 수 있듯이 많은 3.X 삭제 항목들은 직접적으로 대체되었으며, 대체 항목 중 일부는 2.6, 2.7에서도 사용 가능하여 향후 3.X로의 마이그레이션을 지원할 수 있다.

표 C-2 이 책에 영향을 주는 파이썬 3.0에서의 삭제 항목들

삭제 항목	대체 항목	관련 챕터
reload(M)	imp.reload(M)(또는 exec)	3, 23
apply(f, ps, ks)	f(*ps, **ks)	18
`X`	repr(X)	5
X <> Y	X != Y	5
long	int	5
9999L	9999	5
D.has_key(K)	K in D(또는 D.get(key) != None)	8
raw_input	input	3, 10
old input	eval(input( ))	3
xrange	range	13, 14
file	open(그리고 io 모듈 클래스)	9
X.next	X.__next__, next(X)에 의해 호출됨	14, 20, 30
X.__getslice__	슬라이스 객체를 전달받는 X.__getitem__	7, 30
X.__setslice__	슬라이스 객체를 전달받는 X.__setitem__	7, 30
reduce	functools.reduce(또는 루프 코드)	14, 19
execfile(filename)	exec(open(file name).read( ))	3
exec open(filename)	exec(open(file name).read( ))	3
0777	0o777	5
print x, y	print(x, y)	11
print >> F, x, y	print(x, y, file = F)	11
print x, y,	print(x, y, end = ' ')	11
u'ccc' (back in 3.3)	'ccc'	4, 7, 37
'bbb' for byte strings	b'bbb'	4, 7, 9, 37
raise E, V	raise E(V)	33, 34, 35
except E, X:	except E as X:	33, 34, 35

표 C-2 **이 책에 영향을 주는 파이썬 3.0에서의 삭제 항목들 (계속)**

삭제 항목	대체 항목	관련 챕터
def f((a, b)):	def f(x): (a, b) = x	11, 18, 20
file.xreadlines	for line in file:(또는 X = iter(file))	13, 14
D.keys( ), etc. as lists	list(D.keys( ))(딕셔너리 뷰)	8, 14
map( ), range( ), etc. as lists	list(map( )), list(range( ))(내장)	14
map(None, ...)	zip(또는 결과를 보완하기 위한 수동 코드)	13, 20
X = D.keys( ); X.sort( )	sorted(D)(또는 list(D.keys( )))	4, 8, 14
cmp(x, y)	(x > y) - (x < y)	30
X.__cmp__(y)	__lt__, __gt__, __eq__, etc.	30
X.__nonzero__	X.__bool__	30
X.__hex__, X.__oct__	X.__index__	30
Sort comparison functions	Use key = transform or reverse = True	8
Dictionary <, >, <=, >=	Compare sorted(D.items( ))(또는 루프 코드)	8, 9
types.ListType	list(타입은 내장된 이름이 아닌 경우에 한하여 사용)	9
__metaclass__ = M	class C(metaclass = M):	29, 32, 40
__builtin__	builtins(이름 변경)	17
Tkinter	tkinter(이름 변경)	18, 19, 25, 30, 31
sys.exc_type, exc_value	sys.exc_info( )[0], [1]	35, 36
function.func_code	function.__code__	19, 39
__getattr__; 내장된 동작에 의해 실행	포장 클래스에서 __X__ 메서드를 재정의	31, 38, 39
-t, –tt 명령 라인 스위치	탭/공백 사용의 불일치는 항상 에러 발생	10, 12
from ... *, 함수 내 사용 가능	파일 최상위에서만 등장하도록	23
import mod, 동일 패키지에서	from . import mod, 패키지 상대 경로 형태	24
class MyException:	class MyException(Exception):	35
예외 모듈	내장된 범위, 라이브러리 매뉴얼	35
thread, Queue 모듈	_thread, queue(둘 모두 이름 변경)	17
anydbm 모듈	dbm(이름 변경)	28
cPickle 모듈	_pickle(이름 변경. 자동으로 사용됨)	9
os.popen2/3/4	subprocess.Popen(os.popen 유지)	14
문자열 기반의 예외	클래스 기반의 예외(2.6 동일 적용)	33, 34, 35
문자열 모듈 함수	문자열 객체 메서드	7
언바운드 메서드	함수(인스턴스를 통해 호출하는 통계 메서드)	31, 32
혼합된 타입 간의 비교, 정렬	혼합된 비수치형 타입의 비교(또는 정렬)는 에러	5, 9

세 번째 개정판에서의 파이썬 변경 내역: 2.3, 2.4, 2.5

이 책의 세 번째 개정판은 파이썬 2.5와 2003년 후반에 출간된 두 번째 판본 이후에 일어난 언어의 변경 내역 모두를 반영하기 위하여 완전히 개정되었다(두 번째 개정판은 대체로 파이썬 2.2에 근간을 두었으며, 프로젝트 마지막에 접목된 일부 2.3의 특징에 대하여 다루고 있다). 또한, 곧 등장할 대망의 파이썬 3.0의 변경 사항들에 대하여 적절한 시점에 간단히 논하였다. 여기서는 이 책에서 새롭게 또는 기존 내용에 확장하여 다룬, 일부 언어 관련 주요 주제들에 대하여 정리했다(장 번호는 이번 다섯 번째 개정판의 장 번호다).

- B if A else C 조건문 표현(12장, 19장)
- with/as 콘텍스트 매니저(34장)
- try/except/finally 통합(34장)
- 상대적 임포트 구문(24장)
- 제너레이터 표현식(20장)
- 신규 제너레이터 함수 특성(20장)
- 함수 데코레이터(32장, 39장)
- 집합 객체 유형(5장)
- 신규 내장 함수: sorted, sum, any, all, enumerate(13장, 14장)
- 10진수 고정 자릿수 객체 유형(5장)
- 파일, 리스트 컴프리헨션, 반복자(14장, 20장)
- 새로운 개발 도구: Eclipse, distutils, unittest, doctest, IDLE 기능 강화, Shed Skin (2장, 36장)

소소한 언어적 변경(예를 들면 True와 False의 광범위한 사용, 예외 상세 내역을 가져오기 위한 sys.exc_info 추가, 문자열 기반 예외 삭제, 문자열 메서드, apply와 reduce 내장 함수)에 대하여 이 책 전반에서 확인할 수 있다. 또한 세 번째 개정판은 두 번째 개정판에서 신규 항목이었던 내용에 대한 설명을 보강하였는데, 여기에 확장 슬라이스[1]와 apply가 포함된 임의 인수 호출 구문이 포함된다.

1 옮긴이 원저자는 이 확장 슬라이스를 three-limit slice로 표현했으나, 파이썬 매뉴얼에서 확장 슬라이스로 표현하고 있으므로 원문을 변경했다. 확장 슬라이스는 슬라이스 표현에 마지막 인자를 추가하여, 슬라이스 연산 시 중간 단계를 건너뛰거나, 음수 값을 지정하여 역순으로 슬라이스할 수 있는 방법을 제공한다.

이전 또는 이후 파이썬 변경 내역

세 번째 이전의 각 판본에서도 파이썬 변경 내역을 담고 있다. 1999~2003년 사이에 나온 처음 두 판본은 파이썬 2.0과 2.2를 다루고 있으며, 이 책들의 전임자 격인 1996년의 《프로그래밍 파이썬》(초판)은 파이썬 1.3을 기반으로 작성되었다. 하지만 이들은 너무 태고적 이야기라 여기에서 언급하지는 않겠다(최소한 컴퓨터 영역에서는 그렇다).

만약 어떻게든 구할 수 있다면, 상세한 내역은 첫 번째와 두 번째 판본을 참조하기 바란다. 미래를 예측할 순 없지만, 얼마나 오랜 세월 동안 건재하였는지를 생각해 볼 때, 이 책에서 강조한 핵심 아이디어들은 미래의 파이썬에도 여전히 적용될 것이다.

APPENDIX

D

실습 문제 해답

파트 1. 시작하기

실습 문제는 3장의 111쪽 "학습 테스트: 파트 1 실습 문제"를 확인하자.

1. **대화형.** 파이썬이 올바르게 설정되었다는 가정하에 대화형은 다음과 같은 형태를 띤다(IDLE이든, 셸 프롬프트든 그 외 여러분이 선호하는 어떤 방식으로도 실행 가능하다).

```
% python
...저작권 정보 표시...
>>> "Hello World!"
'Hello World!'
>>>                # 세션을 종료하거나 창을 닫으려면 Ctrl + D나 Ctrl + Z
```

2. **프로그램.** 여러분의 코드(즉, 모듈) 파일 module1.py와 운영 시스템 셸 대화형은 다음과 같은 형태를 가진다.

```
print('Hello module world!')

% python module1.py
Hello module world!
```

다시 말하지만, 자유롭게 다른 방식으로 실행해 보아도 된다. 파일 아이콘을 클릭하거나, IDLE의 Run ➡ Run Module 메뉴를 사용하는 등 기타 방식을 활용할 수 있다.

3. **모듈.** 다음의 대화형 세션은 모듈 파일을 임포트하여 실행시키는 방법을 보여 준다.

```
% python
>>> import module1
Hello module world!
>>>
```

인터프리터를 종료하거나 재시작하지 않고 모듈을 재실행하기 위해서는 리로드를 해야 한다는 점을 기억하자. 파일을 다른 디렉터리로 옮기고 다시 임포트하는 것에 대한 질문에 트릭이 숨어 있다. 만약 파이썬이 module1.pyc 파일을 원래 디렉터리에 만들었다면, 파이썬은 여러분이 모듈을 임포트할 때 그 파일을 사용한다. 소스 코드(.py) 파일이 파이썬 검색 경로가 아닌 다른 디렉터리로 옮겨졌다고 하더라도 말이다. 만약 파이썬이 소스 파일 디렉터리에 접근하면 .pyc 파일은 자동으로 작성된다. 해당 파일은 모듈의 컴파일된 바이트 코드 버전을 포함하고 있다. 모듈에 대한 더 자세한 내용은 3장을 참조하면 된다.

4. **스크립트.** 플랫폼이 #! 트릭을 지원한다는 것을 가정하면, 해답은 다음과 같을 것이다(#!줄이 여러분 머신의 다른 경로를 지정해야 할 수는 있다). 이 줄은 파이썬 3.3에 탑재되어 설치되는 윈도우 런처에서는 중요하다. 윈도우 런처는 기본 설정값과 함께 이 줄을 해석하여 스크립트를 실행할 파이썬 버전을 선택한다. 자세한 내용과 예제는 부록 B를 확인하면 된다.

```
#!/usr/local/bin/python    (또는 #!/usr/bin/env python)
print('Hello module world!')
% chmod +x module1.py

% module1.py
Hello module world!
```

5. **오류.** 다음의 대화형 세션(파이썬 3.X에서 실행)은 이 실습 문제를 끝냈을 때 만나게 될 에러 메시지를 보여 준다. 여러분은 실제로 파이썬 예외를 유발한 것이다. 기본 예외 처리 방식은 실행하고 있는 파이썬 프로그래밍을 종료하고 에러 메시지와 함께 스택 트레이스 결과를 화면에 출력하는 것이다. 스택 트레이스는 예외가 발생했을 때 여러분이 프로그램 어디에 있었는지를 보여 준다(만약 에러가 발생했을 때 함수 호출 상태였다면, 'Traceback' 부분에서 모든 활성화되어 있는 호출 레벨을 보여 준다). 10장과 파트 7에서는 try문을 사용하여 예외를 잡아내고 이들을 임의로 처리할 수 있음을 배우게 될 것이다. 또한, 파이썬이 특별한 에러 탐지가 필요한 경우에 대비하여 완전한 소스 코드 디버거를 포함하고 있음을 보게 된다. 지금으로서는 파이썬이 프로그래밍 에러가 발생하면 조용히 멈춰 버리지 않고 의미 있는 메시지를 제공한다는 점에 주목하자.

```
% python
>>> 2 ** 500
32733906078961418700131896968275991522166420460430647894832913680961337964046745
548832700923259041571508866841275600710092172565458853930533285275893 76
>>>
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
>>>
>>> spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

6. **브레이크와 순환 구조.** 다음 코드를 입력하면 여러분은 파이썬에서 순환하는 데이터 구조를 만든 것이다.

```
L = [1, 2]
L.append(L)
```

1.5.1 버전 이전의 파이썬에서는 파이썬 프린터가 객체 내의 순환을 탐지할 만큼 충분히 똑똑하지 않았다. 그래서 여러분이 컴퓨터에서 브레이크 키 조합을 누를 때까지 [1, 2, [1, 2, [1, 2, [1, 2,.....의 끝나지 않는 데이터 스트림을 출력하게 된다. 브레이크 키 조합은 기술적으로 기본 메시지를 출력하는 키보드 간섭 예외를 발생시킨다. 파이썬 1.5.1을 시작으로 프린터는 순환 구조를 탐지하기에 충분할 만큼 똑똑해졌고 [[...]]를 대신 출력함으로써 객체 구조에서 반복 구조를 탐지했다는 것을 알려서 영원히 프린트에만 매달리는 상황을 피하게 되었다.

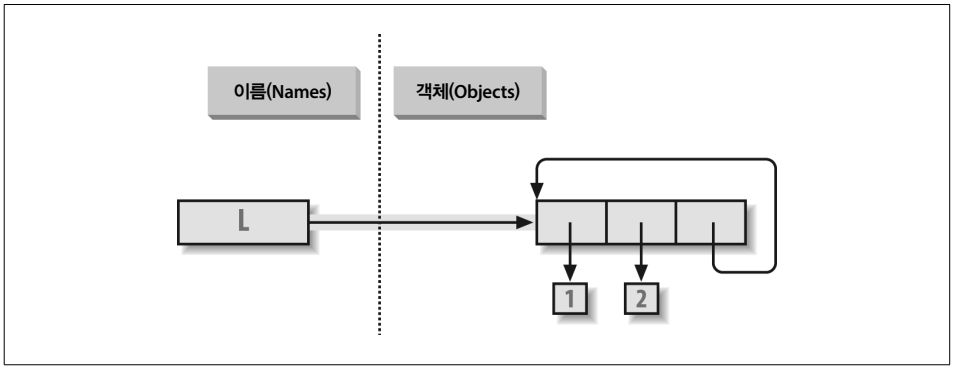

그림 D-1 리스트 자체에 자신을 덧붙임으로써 만들어진 순환 객체. 파이썬은 기본적으로 원래 리스트(리스트의 사본이 아니라)에 대한 참조에 덧붙인다.

이러한 순환의 원인은 알아내기도 어렵고, 파트 2에서 배울 정보들이 필요하므로 여기서 다루는 내용은 미리 보기 정도다. 하지만 요약을 하자면, 파이썬에서의 할당은 항상 객체의 복사본이 아닌 객체에 대한 **참조**를 생성한다. 객체를 메모리와 이를 따르는 포인터로서의 참조의 결집으로 이해하면 된다. 상단의 첫 할당을 실행하면, 이름 L은 두 개의 아이템으로 이루어진 리스트 객체에 대한 명명된 참조가 된다. 이것은 메모리 조각에 대한 포인터다. 파이썬 리스트는 실제로 객체 참조의 배열로, append 메서드를 이용하여 다른 객체 참조를 말미에 고정함으로써 배열을 변경할 수 있다. 여기 append 호출은 L의 앞부분에 대한 참조를 L의 뒷부분에 추가하여 그림 D-1과 같은 순환 구조를 만들게 된다. 리스트 마지막의 포인터가 역으로 리스트 앞부분을 가리키게 되는 것이다.

특별하게 출력되는 것 외에 6장에서 배웠듯이 순환 객체는 파이썬 가비지 콜렉터에 의해 특별하게 처리되어야 하는데, 그렇지 않으면 그 공간은 더 이상 사용하지 않음에도 다른 것들이 할당될 수 없는 불모지로 남게 된다. 현실에서 드물긴 하지만, 임의의 객체 또는 구조를 금하는 일부 프로그램에서는 이러한 반복을 피하기 위하여 여러분이 어디에 있었는지에 대한 기록을 유지하며, 이러한 순환 구조를 직접 찾아내야 할 수도 있다. 믿거나 말거나, 순환 데이터 구조는 이런 특이한 출력에도 불구하고 때로는 유용한 경우가 있다.

파트 2. 타입과 연산

실습 문제는 9장의 395쪽 "학습 테스트: 파트 2 실습 문제"를 확인하자.

1. **기본.** 여기에 각각의 실행 결과와 함께 그 의미에 대하여 간략하게 설명하였다. 다시 말하지만 ;는 한 줄에 하나 이상의 문장을 입력할 때 사용되며(;는 문장 구분자), 콤마는 괄호 안에 표현된 튜플을 구현한다. 또한 / 나눗셈의 결과는 2.X와 3.X의 가장 다른 점이며(상세 내용은 5장 참조), list로 딕셔너리 메서드 호출을 감싸는 것이 3.X에서는 결과를 표현하기 위해서 필요하지만, 2.X는 그렇지 않다는 점을 기억하자(8장 참조).

```
# 수치 데이터

>>> 2 ** 16                              # 2의 16승
65536
>>> 2 / 5, 2 / 5.0                       # 정수 / 2.X에서는 나머지를 버리지만, 3.X는 버리지 않음
(0.40000000000000002, 0.40000000000000002)

# 문자열
```

```
>>> "spam" + "eggs"                    # 문자열 합치기
'spameggs'
>>> S = "ham"
>>> "eggs " + S
'eggs ham'
>>> S * 5                              # 반복
'hamhamhamhamham'
>>> S[:0]                              # 앞에 빈 슬라이스 -- [0:0]
''                                     # 슬라이스된 객체와 동일한 타입이 없다

>>> "green %s and %s" % ("eggs", S)    # 포맷팅
'green eggs and ham'
>>> 'green {0} and {1}'.format('eggs', S)
'green eggs and ham'

# 튜플

>>> ('x',)[0]                          # 단일 아이템 튜플 인덱싱하기
'x'
>>> ('x', 'y')[1]                      # 두 아이템 튜플 인덱싱하기
'y'

# 리스트

>>> L = [1,2,3] + [4,5,6]              # 리스트 연산
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1,2,3]+[4,5,6])[2:4]
[3, 4]
>>> [L[2], L[3]]                       # 오프셋으로부터 가져와서 리스트에 저장함
[3, 4]
>>> L.reverse(); L                     # 메서드: 리스트 순서를 역으로 만들어줌
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                        # 메서드: 리스트를 정렬함
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                         # 메서드: 처음부터 네 번째 아이템 찾기
3

# 딕셔너리

>>> {'a':1, 'b':2}['b']                # 딕셔너리를 키로 인덱싱함
2
>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                         # 새 아이템 생성
>>> D['x'] + D['w']
1
>>> D[(1,2,3)] = 4                     # 튜플이 키로 사용됨(변경 불가)

>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}
```

```
>>> list(D.keys()), list(D.values()), (1,2,3) in D       # 메서드, 키 테스트
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], True)

# 빈 객체들

>>> [[]], ["",[],(),{},None]           # 아무것도 아닌 것들: 빈 객체들
([[]], ['', [], (), {}, None])
```

2. **인덱싱과 슬라이싱.** 경계를 벗어난 인덱싱(예 L[4])은 에러를 일으킨다. 파이썬은 항상 모든 오프셋이 시퀀스의 경계 안에 위치하는지에 대하여 확인한다. 반면에 경계를 벗어난 슬라이싱(예 L[-1000:100])은 문제없이 동작하는데 이는 파이썬이 경계를 넘어가는 슬라이스의 크기를 조정하여 항상 경계에 맞추기 때문이다(경계는 0과 시퀀스 길이로 정해진다). L[3:1]처럼 아래쪽 경계를 위쪽 경계보다 크게 하여 시퀀스를 거꾸로 추출하는 것은 불가능하다. 이렇게 하면 빈 슬라이스가 반환되는데, 파이썬이 슬라이스 경계를 확장하여 아래쪽 경계가 언제나 위쪽 경계보다 작거나 같도록 하기 때문이다. 예를 들어, L[3:1]은 파이썬에 의해 L[3:3]으로 변경된다. 파이썬 슬라이스는 늘 왼쪽에서 오른쪽으로 추출되며, 음의 인덱스를 사용해도 동일하다(음의 인덱스인 경우, 가장 처음 시퀀스의 길이를 더하여 양의 인덱스로 변경하는 작업부터 시작한다). 파이썬 2.3의 3-경계(three-limit) 슬라이스는 이러한 동작 패턴을 약간 변경한다. 예를 들어, L[3:1:-1]은 오른쪽에서 왼쪽으로 추출한다.

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. **인덱싱과 슬라이싱, 그리고 삭제.** 여러분의 인터프리터를 사용한 대화형 세션에서는 아래의 코드와 같은 형태를 띄게 될 것이다. 공백 리스트를 오프셋에 할당하는 것은 거기에 공백 리스트 객체를 저장하는 것이지만, 공백 리스트를 슬라이스에 할당하는 것은 그 슬라이스를 삭제하는 것임을 주의해야 한다. 슬라이스 할당은 다른 시퀀스를 기대하거나, 타입 오류가 난다. 이것은 아이템을 할당된 시퀀스 안에 삽입하지, 시퀀스 자체를 대체하지 않는다.

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
[1, 2, [], 4]
>>> L[2:3] = []
>>> L
[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
>>> L[1:2] = 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

4. **튜플 할당.** X와 Y의 값이 바뀐다. 할당 기호인 =의 좌우에 튜플이 등장하면, 파이썬은 오른쪽 객체를 그 위치에 대응하는 왼쪽 대상에 할당한다. 왼쪽의 대상은 튜플로 보일 수 있지만 실제로는 튜플이 아니며, 단지 독자적인 할당 대상이라는 점을 주목하면 이해하기 쉬울 것이다. 오른쪽에 등장하는 아이템은 튜플이어서 할당 시 풀린다(튜플은 값 교환을 위해 필요한 임시 할당을 제공한다).

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
>>> X
'eggs'
>>> Y
'spam'
```

5. **딕셔너리 키.** 불변 객체라면 어떤 것(정수, 튜플, 문자열 등)이든 딕셔너리 키로 사용될 수 있다. 비록 이 키들 중 일부는 정수형 오프셋으로 보인다 하더라도, 이것은 실제로 딕셔너리다. 혼합형 타입의 키도 정상적으로 동작한다.

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. **딕셔너리 인덱싱.** 존재하지 않는 키를 인덱싱하는 것(D['d'])은 오류를 일으킨다. 존재하지 않는 키에 할당하는 것(D['d'] = 'spam')은 새로운 딕셔너리 아이템을 만든다. 반면에, 리스트에서 경계를 벗어난 인덱싱은 에러를 일으키고, 경계를 벗어난 할당 또한 동일하게 에러를 발생시킨다. 변수명은 딕셔너리 키와 같이 동작한다. 이들은 참조될 때 이미 할당되어 있어야 하지만, 처음 할당될 때에야 생성된다. 실제로 변수명은 원한다면 딕셔너리 키로 처리될 수 있다(이들은 모듈 네임스페이스 또는 스택 프레임 딕셔너리에서 보인다).

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D['a']
1
>>> D['d']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: d
>>> D['d'] = 4
>>> D
{'b': 2, 'd': 4, 'a': 1, 'c': 3}
>>>
>>> L = [0, 1]
>>> L[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

7. **일반 연산.**

- + 연산자는 데이터 타입이 서로 다르거나 혼재되어 있는 경우 사용할 수 없다(예 문자열 + 리스트, 리스트 + 튜플).
- 딕셔너리에서는 +가 동작하지 않는데, 딕셔너리는 시퀀스가 아니기 때문이다.
- append 메서드는 리스트에서만 동작하며, 문자열에서는 동작하지 않는다. 그리고 keys는 딕셔너리에서만 동작한다. append는 그 대상이 제자리에서 확장이 가능하기 때문에 변경 가능하다고 가정한다. 문자열은 변경할 수 없다.
- 슬라이싱과 결합(concatenation)은 항상 처리된 객체와 동일한 타입의 새로운 객체를 반환한다.

```
>>> "x" + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> {} + {}
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for +
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: attribute-less object
>>>
>>> list({}.keys())                    # 3.X에서는 list( )가 필요하나, 2.X에서는 필요 없음
[]
>>> [].keys()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: keys
>>>
>>> []][:]
[]
>>> ""[:]
''
```

8. **문자열 인덱싱.** 약간의 속임수 같은 질문이다. 문자열은 하나의 캐릭터 문자열의 집합체이기 때문에 문자열에 인덱스할 때마다 다시 인덱싱할 수 있는 문자열로 돌아오게 된다. S[0][0][0][0][0]은 첫 번째 문자에 계속해서 인덱싱한다. 이는 일반적으로 리스트가 문자열을 포함하지 않는 한, 리스트에서는 동작하지 않는다(리스트는 임의의 객체를 수용할 수 있다).

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. **불변 타입.** 다음의 두 해결책 모두 답이 된다. 인덱스 할당은 제대로 동작하지 않는데, 문자열은 변경될 수 없기 때문이다.

```
>>> S = "spam"
>>> S = S[0] + 'l' + S[2:]
>>> S
'slam'
```

```
>>> S = S[0] + 'l' + S[2] + S[3]
>>> S
'slam'
```

(37장의 파이썬 3.X와 2.6+의 bytearray 문자열 타입을 함께 참조하자. 가변의 작은 정수들의 시퀀스로, 근본적으로 문자열과 동일하게 처리된다.)

10. **중첩.** 여기 샘플이 있다.

```
>>> me = {'name':('John', 'Q', 'Doe'), 'age':'?', 'job':'engineer'}
>>> me['job']
'engineer'
>>> me['name'][2]
'Doe'
```

11. **파일.** 여기 파이썬에서 텍스트 파일을 만들고 다시 읽어 들이는 방법을 소개한다(ls는 유닉스 명령어로, 윈도우에서는 dir을 사용하면 된다).

```
# maker.py 파일
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')     # 또는: open( ).write( )
file.close()                          # close문이 늘 필요하진 않다.

# reader.py 파일
file = open('myfile.txt')             # 'r' 이 기본 open 모드임
print(file.read())                    # 또는 print(open( ).read( ))

% python maker.py
% python reader.py
Hello file world!

% ls -l myfile.txt
-rwxrwxrwa   1 0        0            19 Apr 13 16:33 myfile.txt
```

파트 3. 문과 구문

실습 문제는 15장의 587쪽 "학습 테스트: 파트 3 실습 문제"를 확인할 것

1. **기본 루프문 작성.** 이 실습을 통해 작업하면 결국 다음과 같은 코드로 마무리될 것이다.

```
>>> S = 'spam'
>>> for c in S:
...     print(ord(c))
...
115
112
97
109

>>> x = 0
>>> for c in S: x += ord(c)           # 또는 x = x + or(c)
...
>>> x
433

>>> x = []
>>> for c in S: x.append(ord(c))
...
>>> x
[115, 112, 97, 109]

>>> list(map(ord, S))                 # 3.X에서는 list( )가 필요하나, 2.X에서는 필요 없음
[115, 112, 97, 109]
>>> [ord(c) for c in S]               # map과 listcomps 리스트 생성을 자동화함
[115, 112, 97, 109]
```

2. **역슬래시 문자.** 예제는 벨 문자(\a)를 50회 출력한다. 여러분 머신에서 처리할 수 있고 IDLE 외부에서 실행된다면 연속된 경고음을 듣게 될지도 모른다(또는 머신이 충분히 빠르다면, 하나의 지속된 음으로 들릴 수도 있겠다). 똑똑. 난 경고했다구요!

3. **딕셔너리 정렬하기.** 여기 이 실습을 통해 작업하는 한 가지 방법(만약 이해되지 않는다면 8장과 14장 참조)을 보여 준다. 여기서 keys와 sort 호출을 반드시 나누어야 한다는 점을 반드시 기억하자. 이는 sort가 None을 반환하기 때문이다. 파이썬 2.2와 그 이후 버전에서는 keys를 호출하지 않고도 직접 딕셔너리 키를 반복할 수 있다(C:\> for key in D:). 하지만 keys 리스트는 이 코드에 의해서 정렬되지 않을 것이다. 좀 더 최근의 파이썬에서는 내장 함수인 sorted로 동일한 효과를 얻을 수 있다.

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = list(D.keys())             # list( ): 3.X에서는 필요하나, 2.X에서는 필요 없음
>>> keys.sort()
>>> for key in keys:
...     print(key, '=>', D[key])
```

```
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7

>>> for key in sorted(D):              # 최근 파이썬에서는 더 나은 방법
...     print(key, '=>', D[key])
```

4. **대안적 프로그램 로직.** 해결책으로 예제 코드는 다음과 같다. 단계 e에서는 2 ** X의 결과를 단계 a와 단계 b 루프 밖의 변수에 할당하고 이를 루프 안에서 사용한다. 결과는 조금 다를 수 있다. 이 실습은 거의 코드의 대안을 가지고 여러모로 활용하기 위해 디자인된 것으로, 논리적이기만 하면 만점을 받을 수 있다.

```
# a

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('at index', i)
        break
    i += 1
else:
    print(X, 'not found')

# b

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'was found at', L.index(p))
        break
else:
     print(X, 'not found')

# c

L = [1, 2, 4, 8, 16, 32, 64]
X = 5
```

```
if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# d

X = 5
L = []
for i in range(7): L.append(2 ** i)
print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# f

X = 5
L = list(map(lambda x: 2**x, range(7)))      # 또는 [2**x for x in range(7)]
print(L)                                     # 3.X에서 모두 출력하려면 list( ), 2.X는 아님

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')
```

5. **코드 유지보수.** 정해진 답이 있는 것은 아니다. 하나의 예로 이 코드에 대하여 내가 작성한 것을 보려면 이 책의 예제 패키지에서 mypydoc.py를 참조하면 된다.

파트 4. 함수와 제너레이터

실습 문제는 21장의 830쪽 "학습 테스트: 파트 4 실습 문제"를 확인하자.

1. **기본.** 여기에 더할 내용은 없지만 print(즉, 여러분이 작성한 함수)를 사용하는 것은 기술적으로 다형적인 연산으로, 각 객체의 타입에 따라 적절한 작업을 수행한다는 점을 기억하자.

```
% python
>>> def func(x): print(x)
...
>>> func("spam")
spam
>>> func(42)
42
```

```
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'food': 'spam'})
{'food': 'spam'}
```

2. **인수.** 아래는 답안 예시다. 기억할 것은 파일은 대화형 세션에서 코딩하는 것과는 달라서, 시험 호출에 대한 결과를 보려면 print문을 사용해야 한다는 점이다. 파이썬은 일반적으로 파일의 표현식 결과를 돌려주지 않는다.

```
def adder(x, y):
    return x + y

print(adder(2, 3))
print(adder('spam', 'eggs'))
print(adder(['a', 'b'], ['c', 'd']))

% python mod.py
5
spameggs
['a', 'b', 'c', 'd']
```

3. **가변 길이의 인수(varargs).** 아래의 파일 adders.py에서는 adder 함수의 두 가지 대안을 확인할 수 있다. 여기에서 어려운 것은 어떻게 하면 어떤 데이터 타입이 전달되더라도 어큐뮬레이터(accumulator)를 빈 값으로 초기화할 것인가라는 문제를 해결하는 것이다. 첫 번째 해답은 타입을 수동으로 검사하여 정수인지 확인하고, 만약 정수가 아니라면 첫 번째 인수의(시퀀스라 가정한다) 공백 슬라이스를 활용한다. 두 번째 해답에서는 첫 번째 인수를 이용해 초기화하고 두 번째부터 그 이후의 인수들을 스캔하는데, 이것은 18장에서 다루었던 min 함수의 변형과 매우 유사하다. 결론은 두 번째 해답이 더 좋은 해결책이다. 이들 모두 모든 인수가 동일한 타입을 가지며, 딕셔너리에서는 동작하지 않는다는 것을 가정한다(파트 2에서 보았듯이, +는 데이터 타입이 여러 개 혼재되어 있거나 딕셔너리일 경우는 동작하지 않는다). 여기에 타입 테스트와 딕셔너리를 허용하는 특수 코드를 추가할 수도 있다. 하지만 이것은 가산점에 해당한다.

```
def adder1(*args):
    print('adder1', end=' ')
    if type(args[0]) == type(0):      # 정수?
        sum = 0                       # 0으로 초기화
    else:                             # 아니라면 시퀀스
        sum = args[0][:0]             # arg1 공백 슬라이스 사용
    for arg in args:
        sum = sum + arg
    return sum
```

```
def adder2(*args):
    print('adder2', end=' ')
    sum = args[0]                       # arg1로 초기화
    for next in args[1:]:
        sum += next                     # 아이템 2..N까지 더하기
    return sum

for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('spam', 'eggs', 'toast'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))

% python adders.py
adder1 9
adder1 spameggstoast
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 spameggstoast
adder2 ['a', 'b', 'c', 'd', 'e', 'f']
```

4. **키워드.** 여기에 파일 mod.py에 코딩된 실습 문제의 첫 번째, 두 번째 파트의 해결책을 제시한다. 키워드 인수로 반복하려면 함수 헤더에서 **args 형태를 사용하고 루프문(예 for x in args.keys(): use args[x])을 사용하거나 args.values()를 사용함으로써 *args의 위치가 지정된 인수를 더하는 것과 동일한 결과를 얻을 수 있다.

```
def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly

print(adder())
print(adder(5))
print(adder(5, 6))
print(adder(5, 6, 7))
print(adder(ugly=7, good=6, bad=5))

% python mod.py
6
10
14
18
18

# 두 번째 부분 해답

def adder1(*args):                      # 위치가 지정된 인수 중 숫자는 모두 합계를 냄
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot
```

```
def adder2(**args):                     # 키워드 인수 중 숫자는 모두 합계함
    argskeys = list(args.keys())        # 3.X에서는 리스트가 필요함
    tot = args[argskeys[0]]
    for key in argskeys[1:]:
        tot += args[key]
    return tot

def adder3(**args):                     # 동일. 하지만 값의 리스트로 변경함
    args = list(args.values())          # 3.X에서는 리스트가 필요함
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):                     # 동일. 하지만 위치가 지정된 변수 버전 재사용
    return adder1(*args.values())

print(adder1(1, 2, 3), adder1('aa', 'bb', 'cc'))
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb', c='cc'))
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb', c='cc'))
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb', c='cc'))
```

5. (와 6) **딕셔너리 도구.** 여기에 실습 문제 **5**와 **6**(파일 dicts.py)에 대한 해답을 제시한다. 이것이 단지 코딩 실습일 뿐인 이유는, 파이썬 1.4에서 딕셔너리 메서드인 D.copy( )와 D1.update(D2)가 추가되어 이를 사용하여 딕셔너리를 복사하고 추가(병합)하는 등의 일들을 처리할 수 있기 때문이다. dict.update에 대한 예제는 8장에서 확인할 수 있으며, 더 자세한 내용은 파이썬 라이브러리 매뉴얼이나 오라일리의 **파이썬 포켓 레퍼런스**를 참조하면 된다. X[:]는 딕셔너리에서는 동작하지 않는데, 이는 딕셔너리는 시퀀스가 아니기 때문이다(자세한 내용은 8장 참조). 또한 복사 대신에 할당(e = d)을 했다면, 이는 **공유된** 딕셔너리 객체에 참조를 생성한 것으로 d의 변경은 e를 변경하는 결과를 낳게 된다.

```
def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new

% python
```

```
>>> from dicts import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1: 1}
>>> y = {2: 2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}
```

6. 5번 참조

7. **인수 맞추기 예제.** 여기 여러분이 갖게 될 대화형들이 어느 인수와 맞춰지는지에 대한 설명과 함께 나열되어 있다.

```
def f1(a, b): print(a, b)              # 일반 인수

def f2(a, *b): print(a, b)             # 위치가 지정된 인수

def f3(a, **b): print(a, b)            # 키워드 가변 길이 인수(keyword varargs)

def f4(a, *b, **c): print(a, b, c)     # 복합형

def f5(a, b=2, c=3): print(a, b, c)    # 기본값 설정

def f6(a, b=2, *c): print(a, b, c)     # 기본값과 위치가 지정된 가변 길이 인수

% python
>>> f1(1, 2)                           # 위치에 의해 맞춰짐(순서가 중요함)
1 2
>>> f1(b=2, a=1)                       # 이름으로 매칭시킴(순서는 중요하지 않음)
1 2

>>> f2(1, 2, 3)                        # 튜플에서 추가 위치 인수를 취합
1 (2, 3)

>>> f3(1, x=2, y=3)                    # 딕셔너리에서 추가 키워드를 취합
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)              # 위 두 종류 모두에 대한 추가 인수 취합
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                              # 나머지 둘은 기본값으로
1 2 3
```

```
>>> f5(1, 4)                          # 마지막 하나만 기본값으로
1 4 3

>>> f6(1)                             # 하나의 인수: 'a'에 매치시킴
1 2 ()
>>> f6(1, 3, 4)                       # 추가 위치가 지정된 인수 취합
1 3 (4,)
```

8. **소수 복습하기.** 여기 소수 예제는 함수와 모듈(파일, primes.py)에 싸여서 여러 번 실행될 수 있다. if 테스트를 추가하여 음수와 0, 1을 잡아낼 수 있도록 하였다. 또한 /를 이번 개정판의 //로 변경하여 이 해답이 파이썬 3.X의 / 진짜 나눗셈(true division)에(5장) 영향을 받지 않도록 하였으며, 부동 소숫점 숫자 또한 지원하도록 했다(2.X에서의 차이점을 확인하려면 from문의 주석 처리를 제거하고 //를 /로 변경하면 된다).

```
#from __future__ import division

def prime(y):
    if y <= 1:                              # 누군가는 y > 1로 시작할 수도
        print(y, 'not prime')
    else:
        x = y // 2                          # 3.X에서는 / 연산 불가
        while x > 1:
            if y % x == 0:                  # 나머지가 없는지?
                print(y, 'has factor', x)
                break                       # 아니면 건너뜀
            x -= 1
        else:
            print(y, 'is prime')

prime(13); prime(13.0)
prime(15); prime(15.0)
prime(3);  prime(2)
prime(1);  prime(-3)
```

여기 실행되는 모듈이 있다. // 연산자는 이 모듈이 부동 소숫점 수에도 작동할 수 있도록 한다.

```
% python primes.py
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
3 is prime
2 is prime
1 not prime
-3 not prime
```

이 함수는 여전히 재활용할 수 없다. 또한 이 함수는 값을 출력하는 대신 값을 반환할 수도 있지만, 실험용으로는 그것만으로도 충분하다. 이는 엄격한 의미의 수학적 소수(부동 소숫점 포함)가 아니며, 여전히 비효율적이다. 보다 수학적 식견을 가진 독자들을 위해 개선할 부분은 실습 문제로 남겨 두도록 하겠다(힌트: range(y, 1, -1)에서 반복되는 for 루프는 while보다 조금 빠르지만 여기에서는 알고리즘이 진짜 병목 지점이다). 대안에 대한 시간 측정을 위하여 파이썬 자체에서 지원하는 timer 또는 표준 라이브러리인 timeit 모듈을 사용하고, 코딩 패턴은 21장의 시간 측정에 대해 다룬 절에서 사용한 패턴과 유사하게 작성하면 된다(해답 10번 함께 확인할 것).

9. **반복과 컴프리헨션.** 다음은 여러분이 작성해야 할 코드다. 여기에 각자 선호하는 코딩 방식에 따라 코드는 달라질 수 있다.

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(map(math.sqrt, values))
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(math.sqrt(x) for x in values)
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. **시간 측정 도구.** 다음은 세제곱근을 구하는 방식에 대한 시간 측정을 위해 작성한 코드를 제시하였다. 이와 함께 CPythons 3.3과 2.7, PyPy 1.9(파이썬 2.7에서 구현한)에서의 결과도 함께 보여 준다. 각 테스트는 세 번의 실행 중 가장 좋은 결과를 취하며, 한 번 실행 시 테스트 함수를 1,000번 호출하는 데 소요되는 총 시간을 측정한다. 각 테스트 함수는 1,000번 반복한다. 각 함수의 마지막 결과는 세 번 모두 동일한 작업을 했는지 검증하기 위하여 출력된다.

```
# timer2.py 파일(2.X and 3.X)
...21장에서 나열한 것과 동일함...

# timesqrt.py 파일
```

```
import sys, timer2
reps = 10000
repslist = range(reps)                  # 2.X를 위해 range list time을 꺼냄

from math import sqrt                   # math.sqrt가 아님: 속성 가져오는 시간을 더함
def mathMod():
    for i in repslist:
        res = sqrt(i)
    return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
    for i in repslist:
        res = i ** .5
    return res

print(sys.version)
for test in (mathMod, powCall, powExpr):
    elapsed, result = timer2.bestoftotal(test, _reps1=3, _reps=1000)
    print ('%s: %.5f => %s' % (test.__name__, elapsed, result))
```

다음은 세 종류의 파이썬에 대한 테스트 결과다. 3.3과 2.7의 결과는 이전 개정판에서 다룬 3.0과 2.6에 비해 대략 두 배 정도 빠르다. 보다 빠른 테스트 머신 덕이다. 테스트에 사용된 각각의 파이썬에서는 math 모듈이 ** 표현식보다 빠른 것으로 보이며, ** 표현식은 pow 호출보다 빠르다. 하지만 여러분은 자신의 머신에서 여러분의 파이썬 버전 위에서 여러분만의 코드로 테스트해 보아야 한다. 또한, 파이썬 3.3은 근본적으로 이 테스트에서 2.7보다 두 배는 느리다는 점에 주목하자. 그리고 PyPy는 부동 소숫점 계산 및 반복을 수행함에도 CPython보다 무려 열 배나 빠르다. 이들 파이썬 중 최근 버전들은 그 결과가 달라질 수 있다. 따라서 스스로 시간을 측정해서 확인해 보자.

```
c:\code> py -3 timesqrt.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
mathMod: 2.04481 => 99.99499987499375
powCall: 3.40973 => 99.99499987499375
powExpr: 2.56458 => 99.99499987499375

c:\code> py -2 timesqrt.py
2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)]
mathMod: 1.04337 => 99.994999875
powCall: 2.57516 => 99.994999875
powExpr: 1.89560 => 99.994999875
```

```
c:\code> c:\pypy\pypy-1.9\pypy timesqrt.py
2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit]
mathMod: 0.07491 => 99.994999875
powCall: 0.85678 => 99.994999875
powExpr: 0.85453 => 99.994999875
```

파이썬 3.X와 2.7의 딕셔너리 컴프리헨션와 이와 동일한 for 루프의 상대적 속도를 측정하기 위해서 다음과 같은 세션을 실행할 수 있다. 파이썬 3.3에서 이 둘은 대략 동일한 속도를 보인다. 하지만 리스트 컴프리헨션과는 다르게 수동 루프가 딕셔너리 컴프리헨션보다 아주 약간 빠르다(이런 차이가 그리 경천동지할 만한 것은 아니다. 결론적으로는 백만 개의 아이템을 가진 50개의 딕셔너리를 만드는 데 겨우 0.5초를 절약했을 뿐이다). 다시 이야기하지만, 이 결과를 온 사방에 복음처럼 전파할 것이 아니라 여러분만의 컴퓨터와 파이썬으로 직접 조사해 보아야 한다.

```
C:\code> c:\python33\python
>>>
>>> def dictcomp(I):
        return {i: i for i in range(I)}

>>> def dictloop(I):
        new = {}
        for i in range(I): new[i] = i
        return new

>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>>
>>> from timer2 import total, bestof
>>> bestof(dictcomp, 10000)[0]              # 10,000개 아이템을 가진 딕셔너리
0.0017095345403959072
>>> bestof(dictloop, 10000)[0]
0.002097576400046819
>>>
>>> bestof(dictcomp, 100000)[0]             # 100,000개 아이템: 열 배 느림
0.012716923463358398
>>> bestof(dictloop, 100000)[0]
0.014129806355413166
>>>
>>> bestof(dictcomp, 1000000)[0]            # 백만 아이템 중 하나: 열 배의 시간이 걸림
0.11614425187337929
>>> bestof(dictloop, 1000000)[0]
0.1331144855439561
```

```
>>>
>>> total(dictcomp, 1000000, _reps=50)[0]     # 총 50개의 백만 아이템 딕셔너리 생성
5.8162020671780965
>>> total(dictloop, 1000000, _reps=50)[0]
6.626680761285343
```

11. **재귀 함수.** 이 함수는 다음과 같이 작성되었다. 단순한 range, 컴프리헨션 타입, 또는 map으로도 여기에 기술한 것과 동일한 작업을 할 수 있지만, 재귀적 호출은 여기에서 실험하기에 충분히 유용하다(print는 3.X에서만 허용되는 함수로, 3.X가 아니라면 __future__로부터 이를 임포트하거나 이와 동일한 작업을 할 수 있는 별도의 코드를 작성하여야 한다).

```
def countdown(N):
    if N == 0:
        print('stop')                  # 2.X: 'stop' 출력
    else:
        print(N, end=' ')              # 2.X: N, 출력
        countdown(N-1)

>>> countdown(5)
5 4 3 2 1 stop
>>> countdown(20)
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 stop

# 비재귀적 방식:
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]

# 3.X에서만:
>>> t = [print(i, end=' ') for i in range(5, 0, -1)]
5 4 3 2 1
>>> t = list(map(lambda x: print(x, end=' '), range(5, 0, -1)))
5 4 3 2 1
```

이 실습에 제너레이터 기반의 해답은 포함시키지 않았으나, 다음에 그중 하나의 예를 제시하였다. 그 외 다른 기법들은 이 경우보다 간단할 것이다. 아마도 제너레이터를 피해야 할 경우에 대한 좋은 예제일 것이다. 제너레이터는 반복이 완료되기 전에는 결괏값을 만들지 않기 때문에 여기에 for 또는 yield from이 필요하다는 점에 주의하자(yield from은 3.3 또는 그 이후 버전에서만 유효하다).

```
def countdown2(N):                             # 제너레이터 함수, 재귀적 용법
    if N == 0:
        yield 'stop'
    else:
        yield N
        for x in countdown2(N-1): yield x  # 3.3+: yield from countdown2(N-1)
```

```
>>> list(countdown2(5))
[5, 4, 3, 2, 1, 'stop']

# 비재귀적 방식
>>> def countdown3():                          # 제너레이터 함수, 단순 용법
        yield from range(5, 0, -1)             # 3.3 이전: for x in range( ): yield x

>>> list(countdown3())
[5, 4, 3, 2, 1]

>>> list(x for x in range(5, 0, -1))          # 제너레이터 표현과 동일
[5, 4, 3, 2, 1]

>>> list(range(5, 0, -1))                     # 제너레이터 아닌 형태와 동일
[5, 4, 3, 2, 1]
```

12. **계승(factorial) 계산하기.** 다음의 파일은 이 실습 문제를 어떻게 코딩하였는지를 보여 준다. 이 파일은 파이썬 3.X와 2.X에서 돌아가고, 3.3에서의 결과물은 파일 마지막의 문자열 리터럴에 주어진다. 이 코드는 다양한 변형이 가능하다. 예를 들어 범위(ranges)는 2..N+1부터 반복에서 벗어날 때까지 실행될 수 있으며, fact2는 lambda를 피하기 위해 reduce(operator.mul, range(N, 1, -1))을 사용할 수 있다.

```
#!python
from __future__ import print_function          # factorials.py 파일
from functools import reduce
from timeit import repeat
import math

def fact0(N):                                  # 재귀 함수
    if N == 1:                                 # 기본적으로 999에서 실패
        return N
    else:
        return N * fact0(N-1)

def fact1(N):
    return N if N == 1 else N * fact1(N-1)     # 재귀 함수, 한 줄 작성

def fact2(N):                                  # 함수
    return reduce(lambda x, y: x * y, range(1, N+1))

def fact3(N):
    res = 1
    for i in range(1, N+1): res *= i           # 반복
    return res

def fact4(N):
```

```
    return math.factorial(N)                    # 표준 라이브러리 "batteries"

# 테스트
print(fact0(6), fact1(6), fact2(6), fact3(6), fact4(6))    # 6*5*4*3*2*1: 모두 720
print(fact0(500) == fact1(500) == fact2(500) == fact3(500) == fact4(500))    # 참

for test in (fact0, fact1, fact2, fact3, fact4):
    print(test.__name__, min(repeat(stmt=lambda: test(500), number=20, repeat=3)))

r"""
C:\code> py -3 factorials.py
720 720 720 720 720
True
fact0 0.003990868798355564
fact1 0.003901433457907475
fact2 0.002732909419593966
fact3 0.002052614370939676
fact4 0.0003401475243271501
"""
```

결론적으로 재귀 호출은 내 파이썬과 머신에서는 가장 느리며, N이 999에 이르렀을 때 sys에서의 스택 사이즈 설정 기본값으로 인해 실행에 실패했다. 19장에 따르면 이 한계는 늘릴 수 있지만, 단순 반복문 또는 표준 라이브러리 도구가 어떤 경우에서도 최고의 방법인 것으로 보인다.

이러한 일반적인 결과가 사실일 경우가 많다. 예를 들어 문자열을 역으로 만드는 데 있어서 재귀적 용법이 가능하다 하더라도, ''.join(reversed(S))이 선호하는 방식일 것이다. 어떻게 그런지를 알아보기 위해 다음 코드의 시간을 측정해 보자. 3.X에서 계승(factorial)에 대해서 말하자면, 재귀적 용법은 오늘날 CPython에서 가장 느리다. 하지만 이 결과가 PyPy에서는 다를 수도 있다.

```
def rev1(S):
    if len(S) == 1:
        return S
    else:
        return S[-1] + rev1(S[:-1])     # 재귀적 용법: 현재 CPython에서 열 배 느림

def rev2(S):
    return ''.join(reversed(S))         # 비재귀적 반복 객체: 단순, 빠름

def rev3(S):
    return S[::-1]                      # 더 나은 방법: 확장 슬라이스로 시퀀스 역으로 바꾸기
```

파트 5. 모듈과 패키지

실습 문제는 25장의 975쪽 "학습 테스트: 파트 5 실습 문제"를 확인하자.

1. **임포트 기본.** 코드 작성을 마무리하면, 여러분의 파일(mymod.py)과 대화형 세션은 아래와 유사한 형태를 가지게 될 것이다. 파이썬은 전체 파일을 문자열의 리스트로 읽어 들일 수 있으며, 내장 함수인 len은 문자열과 리스트의 길이를 반환한다.

```
def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):                                    # 또는 파일 객체를 전달
    return countLines(name), countChars(name)      # 또는 딕셔너리를 반환

% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)
```

여기 카운트 결과에는 주석이나 파일 마지막의 추가 라인들이 포함되거나 또는 안 되었을 수도 있기 때문에 여러분의 카운트 결과는 이와 다를 수도 있다. 이 함수는 전체 파일을 메모리에다 한 번에 올리기 때문에 만약 파일이 여러분 컴퓨터의 메모리로는 수용이 되지 않을 만큼 크다면 동작할 수 없게 된다. 보다 안전하게 만들려면, 반복자를 사용하여 파일을 줄 단위로 읽어 들이며 카운트하는 방식을 쓸 수 있다.

```
def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot

def countChars(name):
    tot = 0
    for line in open(name): tot += len(line)
    return tot
```

제너레이터 표현식으로도 동일한 결과를 얻을 수 있다(하지만 강사는 과도한 마법 사용으로 감점시켜 버릴 수도!).

```
def countlines(name): return sum(+1 for line in open(name))
def countchars(name): return sum(len(line) for line in open(name))
```

유닉스에서는 wc 명령어를 사용하여 결과물을 검증할 수 있다. 윈도우에서는 파일에 오른쪽 마우스 클릭으로 파일의 속성을 볼 수 있다. 또한, 여러분이 작성한 스크립트가 윈도우에서보다 더 적은 글자 수를 결과로 보일 수도 있다. 이식성을 위해 파이썬은 윈도우의 \r\n 줄 끝 표시를 \n으로 바꾸기 때문에 줄당 1바이트(문자)만큼 줄어들게 된다. 윈도우와 정확하게 바이트 카운트를 일치시키려면, 파일을 바이너리 모드('rb')로 열거나 줄 수만큼의 바이트 수를 더해야 한다. 9장과 37장을 살펴보면 텍스트 파일에서 줄 끝 표시의 전환에 대하여 더 자세하게 알아볼 수 있다.

이 실습에서 가장 야심찬 부분(파일 객체를 전달하여 한 번만 파일을 열 수 있게 하는 것)은 여러분이 내장 파일 객체의 seek 메서드를 활용할 것을 요구하는 것이다. 이는 C에서의 fseek 호출과 같은 작업을 한다(무대 뒤에서 이를 호출할지도). seek은 파일에서의 현재 위치를 전달받은 오프셋으로 재설정한다. seek 이후의 입출력 작업은 새로운 위치와 관련되어 일어난다. 파일을 닫고 난 후 다시 열지 않고 해당 파일의 시작 시점으로 되감으려면, file.seek(0)을 호출하면 된다. 파일 read 메서드는 파일에서 현재 위치를 찾아오기 때문에 파일을 다시 읽으려면 되감기를 해야 한다. 이렇게 수정한 내용은 다음과 같다.

```
def countLines(file):
    file.seek(0)                                    # 파일 첫 부분으로 되감기
    return len(file.readlines())

def countChars(file):
    file.seek(0)                                    # 상동(필요시 되감기)
    return len(file.read())

def test(name):
    file = open(name)                               # 파일 객체를 전달
    return countLines(file), countChars(file)       # 파일을 한 번만 염

>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)
```

2. **from/from *.** 여기 from * 부분이 있다. 나머지를 실행하려면 *를 countChars로 교체하면 된다.

```
% python
>>> from mymod import *
>>> countChars("mymod.py")
291
```

3. **__main__.** 적절하게 코드를 작성했다면 이 파일은 프로그램 실행이나 모듈 임포트 중 어떤 모드에서도 동작하게 된다.

```
def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):                                     # 또는 파일 객체 전달
    return countLines(name), countChars(name)       # 또는 딕셔너리 반환

if __name__ == '__main__':
    print(test('mymod.py'))

% python mymod.py
(13, 346)
```

여기가 바로 카운트할 파일 이름을 스크립트에 하드코딩하는 대신, 명령 라인 인수나 사용자 입력으로 받는 방식에 대하여 고려하기 시작하는 시점인 듯하다(sys.argv는 25장에서 상세히 살펴볼 수 있으며, 10장에서는 input(2.X에서는 이 대신 raw_input을 사용함)에 대해 자세히 알아볼 수 있다).

```
if __name__ == '__main__':
    print(test(input('Enter file name:'))          # 콘솔(2.X에서는 raw_input)

if __name__ == '__main__':
    import sys                                    # 명령 라인
    print(test(sys.argv[1]))
```

4. **중첩된 임포트.** 이것은 내가 제시하는 해답이다(파일 myclient.py).

```
from mymod import countLines, countChars
print(countLines('mymod.py'), countChars('mymod.py'))

% python myclient.py
13 346
```

나머지 부분에 대해서 말하자면, mymod 함수는 myclient의 최상위 레벨에서 접근 가능하다(즉 임포트 가능하다). 이는 from이 단순히 임포터에서 이름을 할당하기 때문이다(이는 myclient에서 등장하는 mymod의 def문처럼 동작한다). 예를 들어, 다른 파일에서는 다음과 같이 작성할 수도 있다.

```
import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)
```

만약 myclient가 from 대신에 import를 사용했다면, myclient를 통해 mymod에서 함수를 찾아 들어가기 위한 경로를 사용해야 할 것이다.

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

일반적으로 다른 모듈로부터 모든 이름을 임포트하는 collector 모듈을 정의할 수 있다. 이렇게 되면 단일의 편리한 모듈에서 모든 이름을 확인할 수 있다. 일례로 다음은 somename이라는 이름에 대한 세 개의 서로 다른 복사본을 만드는 부분 코드다(mod1.somename, collector.somename, __main__.somename). 이 세 개 모두 초기에는 동일한 정수 객체를 공유하며, somename이라는 대화형 프롬프트에 다음과 같이 존재한다.

```
# mod1.py 파일
somename = 42

# collector.py 파일
from mod1 import *                    # 여기에 많은 이름을 모음
from mod2 import *                    # from은 이름에 할당함
from mod3 import *

>>> from collector import somename
```

5. **패키지 임포트.** 이 문제를 위해 3번 실습 문제에서 작성한 mymod.py 해답 파일을 디렉터리 패키지에 넣었다. 다음은 윈도우 콘솔 인터페이스에서 디렉터리와 3.3 전까지는 필요한 __init__.py 파일을 설정하는 방법이다. 아마 다른 플랫폼을 위해서는 몇 가지 수정이 필요할 것이다(예를 들어, copy와 notepad 대신에 cp와 vi를 사용한다거나 말이다). 이 작업은 어느 디렉터리에서나 가능하고(나는 본인 코드 디렉터리를 사용하였다) 일부 작업은 파일 탐색기 GUI에서도 할 수 있다.

   작업을 완료하면 __init__.py와 mymod.py를 포함한 mypkg라는 하위 디렉터리가 생길 것이다. 파이썬 3.3의 네임스페이스 패키지가 확장되기 전까지는 mypkg 디렉터리에 (상위 디

렉터리가 아닌) __init__.py 파일이 필요하다. 기술적으로 mypkg는 모듈 검색 경로의 홈 디렉터리에 위치한다. 어떻게 디렉터리 초기화 파일에 작성된 print문이 처음 임포트될 때에만 동작하고, 두 번째부터는 동작하지 않는지를 알아보자. 여기에서도 파일 경로에서 이탈하는 이슈를 피하기 위해 원천 문자열이 사용되었다.

```
C:\code> mkdir mypkg
C:\code> copy mymod.py mypkg\mymod.py
C:\code> notepad mypkg\__init__.py
...print 구문을 작성했음...

C:\code> python
>>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines(r'mypkg\mymod.py')
13
>>> from mypkg.mymod import countChars
>>> countChars(r'mypkg\mymod.py')
346
```

6. **리로드.** 이 문제는 이 책에 등장한 changer.py를 수정해 볼 것을 요구하는 것으로, 여기에서 더 다룰 내용은 없다.

7. **순환 임포트.** 간단히 이야기하자면 recur2를 처음 임포트하는 것은 가능한데, 재귀적 임포트가 이후 recur1에서 일어나지, recure2의 from에서 일어나는 것이 아니기 때문이다.

   풀어서 다시 말하면 처음 recur2를 임포트하는 것은 가능한데, recur1으로부터 recur2로의 임포트는 recur2 전체를, 특정 이름만 가져오는 대신에 가져오기 때문이다. recur2는 recur1으로부터 처음 임포트되었을 때는 불완전하지만, from 대신에 import를 사용하였으므로 안전하다. 파이썬은 이미 생성된 recur2 모듈 객체를 발견, 반환하고 recur1의 나머지 부분을 어떤 결함도 없이 계속 실행하게 된다. recur2의 임포트가 재개되면, 두 번째 from이 recur1의 이름 Y를 발견하기 때문에(완벽하게 실행된다) 어떤 에러도 전달되지 않는다.

   파일을 **스크립트**로 실행하는 것은 모듈로서 이를 임포트하는 것과는 다르다. 이들의 경우는 스크립트의 첫 import 또는 from을 대화형으로 실행했을 때도 동일하다. 예를 들어, recur1을 스크립트로 실행하는 것은 recur2가 대화형에서 임포트되는 것과 동일하기 때문에 recur2가 recur1에서 임포트된 첫 모듈인 것처럼 동작한다. recur2를 스크립트로 실행하는 것은 같은 이유로 실패한다. 이것은 첫 임포트를 대화형으로 실행하는 것과 같다.

파트 6. 클래스와 객체 지향 프로그래밍

실습 문제는 32장의 1350쪽 "학습 테스트: 파트 6 실습 문제"를 확인할 것

1. **상속.** 여기, 실습 문제에 대한 해답 코드를 일부 대화형 테스트와 함께 제시한다(파일 adder.py). __add__ 오버로드는 슈퍼클래스에 한 번만 등장하고, 이것이 서브클래스에서 타입별 특화된 add 메서드를 적용한다.

```
class Adder:
    def add(self, x, y):
        print('not implemented!')
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                # 또는 서브클래스에?
        return self.add(self.data, other)    # 또는 타입을 반환?

class ListAdder(Adder):
    def add(self, x, y):
        return x + y

class DictAdder(Adder):
    def add(self, x, y):
        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new

% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
In 3.3: TypeError: can only concatenate list (not "ListAdder") to list
Earlier: TypeError: __add__ nor __radd__ defined for these operands
```

에러가 발생한 마지막 테스트에서는 + 연산자 오른쪽에 클래스 인스턴스가 나온다. 이를 고치려면, __radd__ 메서드를 사용하면 되며, 30장의 "연산자 오버로딩"절에 그 방법이 자세히 나와 있다.

만약 인스턴스에 포함되는 값을 줄이고 싶다면, 이 파트의 다른 예제들과 마찬가지로, add 메서드가 단지 하나의 인수를 취하도록 재작성하면 된다(여기에 제시된 코드는 adder2.py다).

```
class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                # 단일 인수 전달
        return self.add(other)               # 왼쪽 인수는 self에
    def add(self, y):
        print('not implemented!')

class ListAdder(Adder):
    def add(self, y):
        return self.data + y

class DictAdder(Adder):
    def add(self, y):
        d = self.data.copy()                 # x 대신 self.data를 사용하는 것으로
        d.update(y)                          # 또는 내장 함수를 사용할 수도
        return d

x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)                                     # [1, 2, 3, 4, 5, 6] 출력

z = DictAdder(dict(name='Bob')) + {'a':1}
print(z)                                     # {'name': 'Bob', 'a': 1} 출력
```

값은 사라지지 않고 객체에 붙어 있기 때문에 단언컨대 이 버전은 더 객체 지향적이다. 또한 한 번 이 점을 이해하면, 아마도 add를 모두 없애고 간단하게 두 개의 서브클래스에서 데이터 타입에 특화된 __add__ 메서드를 정의할 수 있게 될 것이다.

2. **연산자 오버로딩.** 해답 코드(파일 mylist.py)는 30장에서 다룬 연산자 오버로딩 메서드를 사용하고 있다. 초깃값은 변경될 여지가 있으므로 생성자에 복사해 두는 것이 중요하다. 클래스 바깥 어딘가에서 공유될지 모르는 객체의 값을 변경하거나 참조하는 것은 여러분도 아마 원하지 않을 것이다. __getattr__ 메서드는 내부에 싸인 리스트로 호출을 넘겨준다. 이 코드를 2.2 또는 그 이후 버전에서 작성하기 좀 더 쉬운 방법을 확인하려면 32장 1233쪽의 "서브클래싱으로 타입 확장하기"절을 참조하면 된다.

```
class MyList:
    def __init__(self, start):
        #self.wrapped = start[:]                    #복사 시작: 부작용 없음
        self.wrapped = list(start)                  # 여기에는 반드시 리스트가 와야 함
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
    def __getitem__(self, offset):                  # 3.X에서는 슬라이스도 전달 가능
        return self.wrapped[offset]                 # __iter__가 없는 경우, 반복을 위해
    def __len__(self):
        return len(self.wrapped)
    def __getslice__(self, low, high):              # 3.X에서는 무시: __getitem__ 사용
        return MyList(self.wrapped[low:high])
    def append(self, node):
        self.wrapped.append(node)
    def __getattr__(self, name):                    # 다른 메서드: sort/reverse/등
        return getattr(self.wrapped, name)
    def __repr__(self):                             # 캐치콜 디스플레이 메서드
        return repr(self.wrapped)

if __name__ == '__main__':
    x = MyList('spam')
    print(x)
    print(x[2])
    print(x[1:])
    print(x + ['eggs'])
    print(x * 3)
    x.append('a')
    x.sort()
    print(' '.join(c for c in x))

c:\code> python mylist.py
['s', 'p', 'a', 'm']
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 's', 'p', 'a', 'm', 's', 'p', 'a', 'm']
a a m p s
```

여기에서 초기 시작값을 슬라이싱 대신에 list를 호출하여 복사한다는 것이 중요하다. 그렇지 않으면 결과가 리스트가 아닐 수도 있으며, 이 경우 append와 같은 리스트 메서드에 대응할 수 없게 된다(일례로, 문자열을 슬라이싱하면 리스트가 아닌 문자열을 반환한다). 아마도 MyList의 시작값을 슬라이싱으로 복사할 수 있을 것이다. 왜냐하면 이 클래스는 슬라이싱 연산을 오버로드하고 예상되는 리스트 인터페이스를 제공하기 때문이다. 그러나 문자열 같은 객체를 슬라이스 기반으로 복사하는 것은 피해야 한다.

3. 서브클래싱. 다음은 나의 해답이다(mysub.py). 여러분 각자의 해답도 이와 유사할 것이다.

```
from mylist import MyList

class MyListSub(MyList):
    calls = 0                                   # 인스턴스에 의해 공유됨
    def __init__(self, start):
        self.adds = 0                           # 인스턴스마다 다름
        MyList.__init__(self, start)

    def __add__(self, other):
        print('add: ' + str(other))
        MyListSub.calls += 1                    # 클래스 범위 기준 카운터
        self.adds += 1                          # 인스턴스별 카운트
        return MyList.__add__(self, other)

    def stats(self):
        return self.calls, self.adds            # 모두 더하기, 자기만 더하기

if __name__ == '__main__':
    x = MyListSub('spam')
    y = MyListSub('foo')
    print(x[2])
    print(x[1:])
    print(x + ['eggs'])
    print(x + ['toast'])
    print(y + ['bar'])
    print(x.stats())

c:\code> python mysub.py
a
['p', 'a', 'm']
add: ['eggs']
['s', 'p', 'a', 'm', 'eggs']
add: ['toast']
['s', 'p', 'a', 'm', 'toast']
add: ['bar']
['f', 'o', 'o', 'bar']
(3, 2)
```

4. **속성 메서드.** 이 실습 문제와 관련하여 다음과 같이 작업하였다. 파이썬 2.X의 고전 클래스에서 연산자는 __getattr__를 통해 속성을 가져오려고 한다. 연산자가 제대로 동작할 수 있도록 값을 반환해야 한다. 32장과 다른 곳에서도 언급했듯이, 파이썬 3.X에서(그리고 만약 새 형식 클래스를 사용한다면 2.X에서도) __getattr__는 내장된 연산자를 위해 호출되지 않기 때문에 이 표현식들은 전혀 가로채지지 않는다. 새 형식 클래스에서 이와 같은 클래스는 __X__ 연산자 오버로딩 메서드를 명시적으로 제정의해야만 한다. 이에 대해서 더 알아

보려면, 28, 31, 32, 38, 39장을 참조하면 된다. 이것은 많은 코드에 영향을 미칠 수 있다!

```
c:\code> py -2
>>> class Attrs:
        def __getattr__(self, name):
            print('get %s' % name)
        def __setattr__(self, name, value):
            print('set %s %s' % (name, value))
>>> x = Attrs()
>>> x.append
get append
>>> x.spam = 'pork'
set spam pork
>>> x + 2
get __coerce__
TypeError: 'NoneType' object is not callable
>>> x[1]
get __getitem__
TypeError: 'NoneType' object is not callable
>>> x[1:5]
get __getslice__
TypeError: 'NoneType' object is not callable

c:\code> py -3
>>> ...같은 초기화 코드...
>>> x + 2
TypeError: unsupported operand type(s) for +: 'Attrs' and 'int'
>>> x[1]
TypeError: 'Attrs' object does not support indexing
>>> x[1:5]
TypeError: 'Attrs' object is not subscriptable
```

5. **집합 객체.** 여기에서는 여러분이 다루게 될 대화형 세션을 보여 준다. 주석은 어떤 메서드가 호출되었는지를 설명한다. 또한 집합은 오늘날의 파이썬에서는 내장된 타입으로, 아래 내용은 코드 작성 연습을 위한 것일 뿐이다(집합에 대해 더 알아보려면 5장을 참조하자).

```
% python
>>> from setwrapper import Set
>>> x = Set([1, 2, 3, 4])              # __init__ 실행
>>> y = Set([3, 4, 5])

>>> x & y                              # __and__, intersect, 그리고 __repr__
Set:[3, 4]
>>> x | y                              # __or__, union, 그리고 __repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set("hello")                   # __init__이 중복 삭제
>>> z[0], z[-1], z[2:]                 # __getitem__
```

```
('h', 'o', ['l', 'o'])

>>> for c in z: print(c, end=' ')      # __iter__(else __getitem__)[3.X print]
...
h e l o
>>> ''.join(c.upper() for c in z)      # __iter__(else __getitem__)
'HELO'
>>> len(z), z                          # __len__, __repr__
(4, Set:['h', 'e', 'l', 'o'])

>>> z & "mello", z | "mello"
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])
```

다중 피연산자 확장 서브클래스에 대한 나의 해답은 다음 클래스와 같은 형태를 띤다(파일 multiset.py). 이는 원래 집합에서 두 개의 메서드만을 대체하면 된다. 다음 클래스 관련 주석은 이것이 어떻게 작동이 되는지 설명한다.

```
from setwrapper import Set

class MultiSet(Set):
    """
    모든 집합의 이름은 상속하되, 교집합(intersect)과 합집합(union)은 다중 연산자를
    지원하기 위하여 확장함. "self"는 여전히 첫 번째 인수임(이제 *args 인수에 저장됨)
    또한 상속된 &와 | 연산자는 두 개의 인수를 갖는 새로운 메서드를 호출하지만,
    두 개보다 많은 인수를 처리하기 위해서는 표현식이 아닌 메서드 호출을 이용해야 함
    여기에서 교집합은 중복 아이템을 제거하지 않지만 Set 생성자는 제거함
    """
    def intersect(self, *others):
        res = []
        for x in self:                          # 첫 번째 시퀀스 검색
            for other in others:                # others 안의 모든 객체에 대해 확인함
                if x not in other: break        # 각 객체에 아이템이 있는가?
            else:                               # No: 루프 밖으로 빠져나감
                res.append(x)                   # Yes: 아이템을 맨 끝에 더함
        return Set(res)

    def union(*args):                           # self는 args[0]
        res = []
        for seq in args:                        # 모든 인수를 위해
            for x in seq:                       # 모든 노드를 위해
                if not x in res:
                    res.append(x)               # 새로운 아이템을 결과에 더함
        return Set(res)
```

이런 확장 버전을 대화형 세션에서 구현한다면 다음과 같을 것이다. &를 사용하거나 intersect를 호출하여 교집합을 구현할 수 있지만, 셋 또는 그 이상의 피연산자를 수용하기

위해서는 intersect를 호출하여야 한다. &는 이항 연산자다. 또한, 만약 multiset 내의 원본을 참조하는 setwrapper.Set을 사용했다면, 우리는 이 변경을 좀 더 투명하게 할 수 있는 MultiSet을 호출했을 것이다(임포트에서 as절은 원한다면 클래스 이름도 바꿀 수 있다).

```
>>> from multiset import *
>>> x = MultiSet([1, 2, 3, 4])
>>> y = MultiSet([3, 4, 5])
>>> z = MultiSet([0, 1, 2])

>>> x & y, x | y                                # 두 개의 피연산자
(Set:[3, 4], Set:[1, 2, 3, 4, 5])

>>> x.intersect(y, z)                           # 세 개의 피연산자
Set:[]
>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]
>>> x.intersect([1,2,3], [2,3,4], [1,2,3])     # 네 개의 피연산자
Set:[2, 3]
>>> x.union(range(10))                          # MultiSet이 아니어도 동작함
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]

>>> w = MultiSet('spam')                        # 문자열 집합
>>> w
Set:['s', 'p', 'a', 'm']
>>> ''.join(w | 'super')
'spamuer'
>>> (w | 'super') & MultiSet('slots')
Set:['s']
```

6. **클래스 트리 링크.** 여기에서는 리스터 클래스를 변경하고 그 포맷을 보여 주는 테스트를 재실행한다. dir 기반의 버전에서도 동일하게 실행하면 된다. 또한, 트리 클라이머 변형에서 클래스 객체의 포맷을 정할 때도 이와 같이 실행하면 된다.

```
class ListInstance:
    def __attrnames(self):
        ...변경되지 않음...

    def __str__(self):
        return '<Instance of %s(%s), address %s:\n%s>' % (
                            self.__class__.__name__,     # 내 클래스명
                            self.__supers(),             # 내 클래스의 슈퍼클래스
                            id(self),                    # 내 주소
                            self.__attrnames())          # 이름 = 값 형태의 목록
    def __supers(self):
        names = []
        for super in self.__class__.__bases__:           # 클래스의 1단계 상위
            names.append(super.__name__)                 # 이름. str(super)아님
```

```
        return ', '.join(names)

    # 또는 ', '.join(super.__name__ for super in self.__class__.__bases__)

c:\code> py listinstance-exercise.py
<Instance of Sub(Super, ListInstance), address 43671000:
        data1=spam
        data2=eggs
        data3=42
>
```

7. **구성 관계.** 나의 해답은 다음과 같다(파일 lunch.py). 코드 내용별 설명은 주석을 참조하면 된다. 이것은 말을 통한 것보다 파이썬으로 문제를 표현하는 것이 더 쉬운 경우에 대한 예제다.

```
class Lunch:
    def __init__(self):                             # Customer, Employee 생성 및 편입
        self.cust = Customer()
        self.empl = Employee()
    def order(self, foodName):                      # 고객 주문 시뮬레이션 시작
        self.cust.placeOrder(foodName, self.empl)
    def result(self):                               # 고객에게 음식 주문 요청
        self.cust.printFood()

class Customer:
    def __init__(self):                             # 내 음식을 None으로 초기화
        self.food = None
    def placeOrder(self, foodName, employee):       # 직원과 함께 주문 생성
        self.food = employee.takeOrder(foodName)
    def printFood(self):                            # 내 음식 이름 출력
        print(self.food.name)

class Employee:
    def takeOrder(self, foodName):                  # 원하는 이름으로 음식 반환
        return Food(foodName)

class Food:
    def __init__(self, name):                       # 음식 이름 저장
        self.name = name

if __name__ == '__main__':
    x = Lunch()                                     # 셀프 테스트 코드
    x.order('burritos')                             # 실행되면 임포트되지 않음
    x.result()
    x.order('pizza')
    x.result()

% python lunch.py
burritos
pizza
```

8. **동물원 동물 계층.** 여기에서는 파이썬에서 분류학을 위한 코드를 작성하는 방법을 제시한다(파일 zoo.py). 이는 인위적이기는 하나, 전반적인 코딩 패턴은 GUI부터 우주선의 직원 데이터베이스에 이르기까지 수많은 실질적 구조에 적용된다. Animal에서 self.speak 참조는 독자적인 상속 검색을 유발하여 서브클래스에서 speak를 찾게 한다. 실습 문제 설명마다 대화형 세션으로 테스트해 보기 바란다. 새로운 클래스를 가지고 이 계층 구조를 확장한 후, 트리 내에 다양한 클래스 인스턴스를 만들어 보자.

```
class Animal:
    def reply(self):   self.speak()          # 서브클래스로 돌아감
    def speak(self):   print('spam')         # 수정된 메시지

class Mammal(Animal):
    def speak(self):   print('huh?')

class Cat(Mammal):
    def speak(self):   print('meow')

class Dog(Mammal):
    def speak(self):   print('bark')

class Primate(Mammal):
    def speak(self):   print('Hello world!')

class Hacker(Primate): pass                  # Primate로부터 상속
```

9. **죽은 앵무새 촌극.** 파일 parrot.py에서 죽은 앵무새 촌극을 구현했다. 여기서 주목할 것은 Actor 슈퍼클래스에서 line 메서드가 어떻게 동작하는가다. self 속성에 두 번 접근함으로써 파이썬을 인스턴스로 두 번 돌려보내고, 따라서 두 번의 상속 검색을 일으킨다. self.name과 self.says()는 특정 서브클래스에서 정보를 찾는다.

```
class Actor:
    def line(self): print(self.name + ':', repr(self.says()))

class Customer(Actor):
    name = 'customer'
    def says(self): return "that's one ex-bird!"

class Clerk(Actor):
    name = 'clerk'
    def says(self): return "no it isn't..."

class Parrot(Actor):
    name = 'parrot'
    def says(self): return None
```

```
class Scene:
    def __init__(self):
        self.clerk    = Clerk()              # 일부 인스턴스 내포시킴
        self.customer = Customer()           # Scene은 복합 클래스
        self.subject  = Parrot()

    def action(self):
        self.customer.line()                 # 내포된 아이템들에 위임
        self.clerk.line()
        self.subject.line()
```

파트 7. 예외와 도구들

실습 문제는 36장의 1461쪽 "학습 테스트: 파트 7 실습 문제"를 확인하자.

1. **try/except.** 내(저자) 버전의 oops 함수(파일 oops.py)는 다음과 같다. 코딩과 관련 없는 질문에 관해서라면, IndexError 대신에 KeyError를 일으키도록 oops를 변경하는 것은 try 핸들러가 예외를 잡아내지 못한다는 것을 의미한다. 이 예외는 최상위 레벨까지 전파되어 파이썬의 기본 에러 메시지를 유발시킨다. KeyError와 IndexError 이름은 가장 바깥 영역의 내장된 이름 범위로부터 온 것이다('LEGB' 중 B 영역). 스스로 확인해 보려면 3.X에서 builtins(2.X에서는 __builtin__)를 임포트하고 이를 dir 함수에 인수로 전달하면 된다.

```
def oops():
    raise IndexError()

def doomed():
    try:
        oops()
    except IndexError:
         print('caught an index error!')
    else:
        print('no error caught...')

if __name__ == '__main__': doomed()

% python oops.py
caught an index error!
```

2. **예외 객체와 리스트.** 나만의 예외 처리를 위하여 이 모듈을 확장한 코드는 다음과 같다(파일 oops2.py).

```
from __future__ import print_function              # 2.X

class MyError(Exception): pass

def oops():
    raise MyError('Spam!')

def doomed():
    try:
        oops()
    except IndexError:
        print('caught an index error!')
    except MyError as data:
        print('caught error:', MyError, data)
    else:
        print('no error caught...')

if __name__ == '__main__':
    doomed()

% python oops2.py
caught error: <class '__main__.MyError'> Spam!
```

모든 클래스 예외처럼 인스턴스는 as 변수 data에 의해 접근 가능하다. 에러 메시지는 클래스(<...>)와 클래스의 인스턴스(Spam!) 둘 다를 보여 준다. 인스턴스는 파이썬의 Exception 클래스로부터 __init__과 __repr__ 또는 __str__ 모두를 상속해야만 하는데, 그렇지 않으면 클래스처럼 출력할 것이다. 35장을 보면 내장된 예외 클래스가 어떻게 동작하는지 더 알아볼 수 있다.

3. **에러 처리.** 이 문제에 대한 한 가지 해답은 다음과 같다(파일 exctools.py). 나는 대화형 세션 대신 파일로 테스트를 진행했지만, 결과는 확실히 비슷하게 나올 것이다. 여기에서 사용된 공백의 except와 sys.exc_info 접근 방식은 as 변수와 함께 나열된 Exception에서는 잡지 못하는 exit 관련 예외를 잡아낸다. 이는 아마 대부분의 응용 코드에서 이상적이진 않겠지만, 예외 방화벽 같은 역할을 위해 디자인된 도구에서는 유용할 것이다.

```
import sys, traceback

def safe(callee, *pargs, **kargs):
    try:
        callee(*pargs, **kargs)                    # 다른 모든 예외를 캐치하거나
    except:                                        # 또는 'except Exception as E:'
        traceback.print_exc()
        print('Got %s %s' % (sys.exc_info()[0], sys.exc_info()[1]))

if __name__ == '__main__':
```

```
    import oops2
    safe(oops2.oops)

c:\code> py -3 exctools.py
Traceback (most recent call last):
  File "C:\code\exctools.py", line 5, in safe
    callee(*pargs, **kargs)                 # 다른 모든 예외를 캐치함
  File "C:\code\oops2.py", line 6, in oops
    raise MyError('Spam!')
oops2.MyError: Spam!
Got <class 'oops2.MyError'> Spam!
```

다음의 코드는 이것을 어떤 함수가 일으킨 예외라도 잡아내고 감쌀 수 있는 **함수 데코레이터**로 전환할 수 있다. 이에 응용된 기법은 32장에서 소개되었으며, 전반적인 내용은 그다음 파트인 39장에서 다루고 있다. 이는 예외가 명시적으로 전달되는 것보다는 함수의 기능을 보강하는 효과를 가지고 있다.

```
import sys, traceback

def safe(callee):
    def callproxy(*pargs, **kargs):
        try:
            return callee(*pargs, **kargs)
        except:
            traceback.print_exc()
            print('Got %s %s' % (sys.exc_info()[0], sys.exc_info()[1]))
            raise
    return callproxy

if __name__ == '__main__':
    import oops2

    @safe
    def test():
        oops2.oops()

    test()
```

4. **자가학습 예제.** 시간이 허락하는 대로 다음의 예제들을 이용하여 스스로 학습해 보자. 더 많은 내용은 이 예제들을 차용해 온 《프로그래밍 파이썬》과 같은 책과 웹을 참조하면 된다.

```
# 한 디렉터리에서 가장 큰 파이썬 소스 코드 파일 찾기

import os, glob
dirname = r'C:\Python33\Lib'
```

```
allsizes = []
allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])


# 전체 디렉터리 트리에서 가장 큰 파이썬 소스 코드 파일 찾기

import sys, os, pprint
if sys.platform[:3] == 'win':
    dirname = r'C:\Python33\Lib'
else:
    dirname = '/usr/lib/python'

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    for filename in filesHere:
        if filename.endswith('.py'):
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])


# 모듈 임포트 검색 경로에서 가장 큰 파이썬 소스 코드 파일 찾기

import sys, os, pprint
visited = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('skipping', pypath)
                allsizes.append((pysize, pypath))
```

```
allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])


# 콤마로 구분된 텍스트 파일의 컬럼 수 합계 구하기

filename = 'data.txt'
sums = {}

for line in open(filename):
    cols = line.split(',')
    nums = [int(col) for col in cols]
    for (ix, num) in enumerate(nums):
        sums[ix] = sums.get(ix, 0) + num

for key in sorted(sums):
    print(key, '=', sums[key])


# 이전과 동일하지만, 합계를 구하기 위해 딕셔너리 대신 리스트 사용하기

import sys
filename = sys.argv[1]
numcols  = int(sys.argv[2])
totals   = [0] * numcols

for line in open(filename):
    cols = line.split(',')
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums)]

print(totals)


# 일련의 스크립트 결과에서 회귀 테스트

import os
testscripts = [dict(script='test1.py', args=''),     # 또는 전역 스크립트/인수 디렉터리
               dict(script='test2.py', args='spam')]

for testcase in testscripts:
    commandline = '%(script)s %(args)s' % testcase
    output = os.popen(commandline).read()
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Created:', result)
    else:
        priorresult = open(result).read()
        if output != priorresult:
            print('FAILED:', testcase['script'])
            print(output)
```

```
        else:
            print('Passed:', testcase['script'])


# tkinter로 색깔과 폰트 사이즈를 변경할 수 있는 버튼을 가진 GUI 구현하기(2.X에서는 Tkinter)

from tkinter import *                           # 2.X에서는 Tkinter를 사용할 것
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'white', 'cyan', 'purple']

def reply(text):
    print(text)
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color).pack()
    L.config(fg=color)

def timer():
    L.config(fg=random.choice(colors))
    win.after(250, timer)

def grow():
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)

win = Tk()
L = Label(win, text='Spam',
          font=('arial', fontsize, 'italic'), fg='yellow', bg='navy',
          relief=RAISED)
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='press', command=(lambda: reply('red'))).pack(side=BOTTOM, fill=X)
Button(win, text='timer', command=timer).pack(side=BOTTOM, fill=X)
Button(win, text='grow', command=grow).pack(side=BOTTOM, fill=X)
win.mainloop()


# 이전과 유사하나, 클래스를 사용하여 각 창마다 자신만의 상태 정보를 가지도록 구현하기

from tkinter import *
import random

class MyGui:
    """
    색상을 변경하거나 레이블 크기를 변경하는 버튼이 포함된 GUI
    """
    colors = ['blue', 'green', 'orange', 'red', 'brown', 'yellow']

    def __init__(self, parent, title='popup'):
        parent.title(title)
        self.growing = False
```

```
        self.fontsize = 10
        self.lab = Label(parent, text='Gui1', fg='white', bg='navy')
        self.lab.pack(expand=YES, fill=BOTH)
        Button(parent, text='Spam', command=self.reply).pack(side=LEFT)
        Button(parent, text='Grow', command=self.grow).pack(side=LEFT)
        Button(parent, text='Stop', command=self.stop).pack(side=LEFT)

    def reply(self):
        "change the button's color at random on Spam presses"
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color,
                font=('courier', self.fontsize, 'bold italic'))

    def grow(self):
        "start making the label grow on Grow presses"
        self.growing = True
        self.grower()

    def grower(self):
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fontsize, 'bold'))
            self.lab.after(500, self.grower)

    def stop(self):
        "stop the button growing on Stop presses"
        self.growing = False

class MySubGui(MyGui):
    colors = ['black', 'purple']              # 색상 선택을 바꿀 수 있도록 변경

MyGui(Tk(), 'main')
MyGui(Toplevel())
MySubGui(Toplevel())
mainloop()
```

 # 이메일 수신함 탐색 및 유지 관리

```
"""
pop 이메일 박스를 살펴보고, 헤더만 가져온 뒤, 전체 메시지를 다운로드받지 않고도 삭제할 수 있도록
"""

import poplib, getpass, sys

mailserver = 'your pop email server name here'              # pop.server.net
mailuser   = 'your pop email user name here'
mailpasswd = getpass.getpass('Password for %s?' % mailserver)

print('Connecting...')
server = poplib.POP3(mailserver)
server.user(mailuser)
```

```
server.pass_(mailpasswd)

try:
    print(server.getwelcome())
    msgCount, mboxSize = server.stat()
    print('There are', msgCount, 'mail messages, size ', mboxSize)
    msginfo = server.list()
    print(msginfo)
    for i in range(msgCount):
        msgnum  = i+1
        msgsize = msginfo[1][i].split()[1]
        resp, hdrlines, octets = server.top(msgnum, 0)          # 헤더만 가져오기
        print('-'*80)
        print('[%d: octets=%d, size=%s]' % (msgnum, octets, msgsize))
        for line in hdrlines: print(line)

        if input('Print?') in ['y', 'Y']:
            for line in server.retr(msgnum)[1]: print(line)     # 전체 메시지 가져오기
        if input('Delete?') in ['y', 'Y']:
            print('deleting')
            server.dele(msgnum)                         # 서버에서 메시지 삭제하기
        else:
            print('skipping')
finally:
    server.quit()                                       # 메일박스 잠금해제 확인
input('Bye.')                                           # 윈도우에서 창이 위에 뜨도록


# 웹 브라우저와 통신하기 위한 CGI 서버단의 스크립트

#!/usr/bin/python
import cgi
form = cgi.FieldStorage()                               # 데이터로부터 해석(파싱)
print("Content-type: text/html\n")                      # 헤더 아래에 공백 줄 추가
print("<HTML>")
print("<title>Reply Page</title>")                      # HTML 응답 페이지
print("<BODY>")
if not 'user' in form:
    print("<h1>Who are you?</h1>")
else:
    print("<h1>Hello <i>%s</i>!</h1>" % cgi.escape(form['user'].value))
print("</BODY></HTML>")


# 파이썬 객체로 shelve에 데이터를 덧붙이는 데이터베이스 스크립트

# shelve는 28장에서, pickle 예제는 31장에서 확인할 것

rec1 = {'name': {'first': 'Bob', 'last': 'Smith'},
        'job': ['dev', 'mgr'],
        'age': 40.5}

rec2 = {'name': {'first': 'Sue', 'last': 'Jones'},
```

```
        'job': ['mgr'],
        'age': 35.0}

import shelve
db = shelve.open('dbfile')
db['bob'] = rec1
db['sue'] = rec2
db.close()


# 이전 스크립트에서 생성된 shelve를 출력하고 업데이트하는 데이터베이스 스크립트

import shelve
db = shelve.open('dbfile')
for key in db:
    print(key, '=>', db[key])

bob = db['bob']
bob['age'] += 1
db['bob'] = bob
db.close()


# MySql 데이터베이스를 조회하고 데이터를 추가하기 위한 데이터베이스 스크립트

from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='XXXXXXX')
curs = conn.cursor()
try:
    curs.execute('drop database testpeopledb')
except:
    pass                                                    # 존재하지 않음

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay int(4))')

curs.execute('insert people values (%s, %s, %s)', ('Bob', 'dev', 50000))
curs.execute('insert people values (%s, %s, %s)', ('Sue', 'dev', 60000))
curs.execute('insert people values (%s, %s, %s)', ('Ann', 'mgr', 40000))

curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

curs.execute('select * from people where name = %s', ('Bob',))
print(curs.description)
colnames = [desc[0] for desc in curs.description]
while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
    for (name, value) in zip(colnames, row):
```

```
        print('%s => %s' % (name, value))

conn.commit()                                   # 추가된 레코드 저장


# FTP로 파일을 가져오고 열어서 재생함

import webbrowser, sys
from ftplib import FTP                          # 소켓 기반의 FTP 도구
from getpass import getpass                     # 패스워드 입력 숨김 처리
if sys.version[0] == '2': input = raw_input     # 2.X 호환성

nonpassive = False                              # 서버를 위해 FTP 모드를 활성화할 것인가?
filename   = input('File?')                     # 다운로드할 파일
dirname    = input('Dir? ') or '.'              # 파일 가져올 원격 디렉터리
sitename   = input('Site?')                     # 접속할 FTP 사이트
user       = input('User?')                     # 익명일 경우 () 사용
if not user:
    userinfo = ()
else:
    from getpass import getpass                 # 패스워드 입력 숨김 처리
    userinfo = (user, getpass('Pswd?'))

print('Connecting...')
connection = FTP(sitename)                      # FTP 사이트 연결
connection.login(*userinfo)                     # 기본은 익명 로그인
connection.cwd(dirname)                         # 1번에 지역 파일에 1k씩 전달
if nonpassive:                                  # 서버가 필요할 때, FTP 활성화
    connection.set_pasv(False)

print('Downloading...')
localfile = open(filename, 'wb')                # 다운로드 파일을 저장할 지역 파일
connection.retrbinary('RETR ' + filename, localfile.write, 1024)
connection.quit()
localfile.close()

print('Playing...')
webbrowser.open(filename)
```