
Python Frequently Asked Questions

릴리스 3.14.0a7

Guido van Rossum and the Python development team

4월 27, 2025

Contents

1	일반적인 파이썬 FAQ	1
2	프로그래밍 FAQ	7
3	설계와 역사 FAQ	39
4	라이브러리와 확장 FAQ	51
5	확장/내장 FAQ	61
6	윈도우 파이썬 FAQ	67
7	그래픽 사용자 인터페이스 FAQ	71
8	“왜 내 컴퓨터에 파이썬이 설치되어 있습니까?” FAQ	73
A	용어집	75
B	이 설명서에 관하여	93
C	역사와 라이선스	95
D	저작권	115
	색인	117

일반적인 파이썬 FAQ

1.1 일반적인 정보

1.1.1 파이썬이 무엇입니까?

파이썬은 인터프리터 방식의, 해석되고 상호작용이 가능한, 객체 지향적 프로그래밍 언어입니다. 모듈, 예외, 동적 타이핑, 굉장히 높은 수준의 동적 데이터 타입, 그리고 클래스를 내포하고 있습니다. 절차적이나 함수형 프로그래밍과 같은 객체 지향 프로그래밍 이외의 여러 프로그래밍 패러다임을 지원합니다. 파이썬은 굉장히 깔끔한 구문으로 놀랄만한 힘을 결합합니다. 이 언어는 다양한 시스템 호출과 라이브러리 뿐만 아니라 다양한 윈도우 시스템 관련 인터페이스를 갖고 있으며, C 언어와 C++ 언어로 확장이 가능합니다. 또한 프로그래밍 인터페이스가 필요한 애플리케이션의 확장 언어로도 사용할 수 있습니다. 마지막으로, 파이썬은 리눅스와 macOS를 포함하는 많은 유닉스 변종과 윈도우에서 돌아가는 포터블 언어입니다.

추가 정보는 [tutorial-index](#)에 있습니다. [파이썬 초보자 가이드](#)에서 다른 기초 자습서와 파이썬을 배우기 위한 자료들을 볼 수 있습니다.

1.1.2 파이썬 소프트웨어 재단이 무엇입니까?

파이썬 소프트웨어 재단이란 파이썬 2.1과 그 뒤 버전의 저작권을 소유하고 있는 독립적이고 비영리적인 단체입니다. 이 재단의 사명은 파이썬의 사용을 대중화 시키고 파이썬 관련 오픈소스 기술을 발전시키는 것입니다. 파이썬 소프트웨어 재단의 홈페이지: <https://www.python.org/psf/>.

파이썬 소프트웨어 재단으로의 기부는 미국에서 세액공제 적용 대상입니다. 파이썬을 사용하고 도움이 된다고 생각한다면 [PSF 기부 페이지](#)를 통해 기여해 주십시오.

1.1.3 파이썬을 사용할 때 저작권 관련 제한이 있습니까?

본인이 생성한 파이썬에 대해 저작권을 명시하고, 이와 관련된 정보를 어떠한 형태로든 설명서에 표시하면 본인의 소스로 무엇이든 할 수 있습니다. 저작권 관련 규칙을 따르면 파이썬을 상업 목적으로 사용해도 괜찮으며, 소스코드나 (수정된 또는 수정되지 않은) 이진형태의 파이썬을 팔거나, 파이썬을 내장하는 상품을 팔아도 괜찮습니다. 그럼에도 저희는 파이썬이 사용된 모든 상업 활동을 알고 싶습니다.

See the [license page](#) to find further explanations and the full text of the PSF License.

파이썬 로고는 상표로 등록되어 있으며, 일부 경우에는 사용하기 위한 허가가 필요합니다. 더 많은 정보를 원하시면 [상표 사용 정책](#)을 참조하십시오.

1.1.4 파이썬은 애초에 왜 만들어졌습니까?

여기 Guido van Rossum 씨가 작성한 굉장히 요약된 파이썬의 탄생 계기입니다:

저는 CWI(Centrum Wiskunde & Informatica)의 ABC 그룹에서 인터프리터 언어를 도입하며 광범위한 경험을 해왔고, 그 그룹과 일하며 언어의 디자인에 대해 많은 것을 배웠습니다. 들여쓰기로 문단 묶기나 매우-상위-레벨 자료형을 포함한 것과 같은 파이썬의 다양한 기능들이 이 당시 경험을 바탕으로 만들어졌습니다.

ABC 언어에 있어서 저는 불만도 있었지만, 마음에 드는 기능도 많았습니다. ABC 언어(그리고 언어의 이행과정)는 제 불만을 극복할 만큼 확장할 방법이 없었죠. -사실 확장성이 떨어지는 것이 이 언어의 가장 큰 문제점 중 하나였습니다. 저는 Modula-2+를 사용하는데 약간의 경험이 있었고 Modula-3의 디자이너 분들과 대화도 나눠보았으며 Modula-3 관련 레포트도 읽어본 상태였습니다. Modula-3은 예외에서 쓰인 문법과 의미론을 비롯한 일부 파이썬 기능들의 기원입니다.

저는 CWI에서 Amoeba 분산 운영체제 그룹에 속해 일하고 있었습니다. Amoeba는 Bourne 셸로는 쉽게 접근하지 못하는 고유의 시스템 호출 인터페이스를 내장하고 있었기 때문에 저희는 C 프로그래머나 Bourne 셸 스크립트를 작성하는 것이 아닌 더 좋은 시스템 관리 방법이 필요했습니다. Amoeba에서 에러 처리를 해본 경험 덕분에 저는 프로그래밍 언어의 기능에 있어서 예외의 중요성을 인지하고 있었습니다.

ABC의 문법을 지키면서 Amoeba 시스템 호출에 접근 가능한 스크립트 언어가 필요하다는 것을 깨달았죠. Amoeba 특유의 언어를 만드는 건 어리석은 행동이라고 느껴서 대체로 확장 가능한 언어가 필요한 상황이라고 판단했습니다.

1989년 크리스마스 연휴에 저는 많은 시간이 있었고, 그 참에 도전해보기로 했습니다. 다음 해에도 제 개인적인 시간을 투자하며 언어를 완성시켜 가는 와중에 Python은 Amoeba 프로젝트에서 날이 갈수록 성공적으로 사용되고 있었습니다. 그리고 제 동료들의 피드백을 통해 저는 초기 개선사항들을 추가할 수 있었죠.

1991년 2월, 1년이 조금 넘는 개발 기간 후, 저는 USENET에 올리기로 했습니다. 나머지는 Misc/HISTORY 파일에 있습니다.

1.1.5 파이썬의 강점은 무엇입니까?

파이썬은 다양한 상황에 적용 가능한, 범용성 있는 고급언어입니다.

The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for library-index to get an idea of what's available. A wide variety of third-party extensions are also available. Consult [the Python Package Index](#) to find packages of interest to you.

1.1.6 파이썬 버전 관리 규칙은 무엇입니까?

Python versions are numbered “A.B.C” or “A.B”:

- A is the major version number – it is only incremented for really major changes in the language.
- B is the minor version number – it is incremented for less earth-shattering changes.
- C is the micro version number – it is incremented for each bugfix release.

Not all releases are bugfix releases. In the run-up to a new feature release, a series of development releases are made, denoted as alpha, beta, or release candidate. Alphas are early releases in which interfaces aren't yet finalized; it's not unexpected to see an interface change between two alpha releases. Betas are more stable, preserving existing interfaces but possibly adding new modules, and release candidates are frozen, making no changes except as needed to fix critical bugs.

Alpha, beta and release candidate versions have an additional suffix:

- The suffix for an alpha version is “aN” for some small number N.
- The suffix for a beta version is “bN” for some small number N.

- The suffix for a release candidate version is “rcN” for some small number *N*.

In other words, all versions labeled *2.0a**N* precede the versions labeled *2.0b**N*, which precede versions labeled *2.0rc**N*, and *those* precede 2.0.

또한 “+” 접미사가 붙은 버전을 마주할 수도 있습니다. 예) “2.2+”. 이는 CPython 개발 저장소에서 바로 빌드된, 배포되지 않은 버전입니다. 실제로는 이 버전의 최종 소규모 배포가 이루어질 때 다음 소규모 버전으로 증가하여 “a0” 버전이 됩니다. 예) “2.4a0”.

See the [Developer’s Guide](#) for more information about the development cycle, and [PEP 387](#) to learn more about Python’s backward compatibility policy. See also the documentation for `sys.version`, `sys.hexversion`, and `sys.version_info`.

1.1.7 파이썬 소스를 어떻게 얻을 수 있습니까?

최신 파이썬 소스는 python.org 또는 <https://www.python.org/downloads/> 에서 항상 구할 수 있고, 최신 개발 소스는 <https://github.com/python/cpython/> 에서 구할 수 있습니다.

소스 배포판은 모든 C 언어 코드, 스텝크스 형식의 문서, 파이썬 라이브러리 모듈, 예제 프로그램들, 그리고 일부 유용한 무료 배포 소프트웨어를 포함합니다. 소스는 대부분의 유닉스 플랫폼에서 추가 설정 없이 컴파일되고 실행 가능합니다.

소스 코드를 받고 컴파일하는 부분에서 추가 정보를 원하시면 [파이썬 개발자 가이드](#)의 개요를 참고하십시오.

1.1.8 파이썬 설명 문서는 어떻게 구합니까?

현재 파이썬의 안정화 버전에 관련된 표준 문서는 <https://docs.python.org/3/>에서 확인할 수 있습니다. PDF, 텍스트 문서, 그리고 다운로드 가능한 HTML 버전들은 <https://docs.python.org/3/download.html> 에서 구할 수 있습니다.

The documentation is written in reStructuredText and processed by the [Sphinx documentation tool](#). The reStructuredText source for the documentation is part of the Python source distribution.

1.1.9 저는 프로그래밍을 해본 적이 없습니다. 파이썬 튜토리얼이 있습니까?

다양한 튜토리얼과 책이 존재합니다. 표준 문서로는 [tutorial-index](#)가 있습니다.

[초보자 가이드](#)에서 튜토리얼 목록과 같은 초보 파이썬 개발자를 위한 정보를 확인하십시오.

1.1.10 파이썬에 특화된 뉴스 그룹이나 메일링 리스트가 있습니까?

뉴스 그룹은 `comp.lang.python`, 그리고 메일링 리스트는 [python-list](#)에서 확인할 수 있습니다. 뉴스 그룹과 메일링 리스트는 서로 연동되어 있어서 뉴스를 읽는다면 메일을 구독할 필요가 없습니다. `comp.lang.python`은 활발하고 하루에도 수백 개의 포스팅이 올라오기 때문에 Usenet 독자들은 이 매체를 통해 더 원활하게 소통할 수 있을 것입니다.

Announcements of new software releases and events can be found in `comp.lang.python.announce`, a low-traffic moderated list that receives about five postings per day. It’s available as the [python-announce mailing list](#).

<https://www.python.org/community/lists/> 에서 다른 메일링 리스트와 뉴스 그룹에 대한 정보를 확인할 수 있습니다.

1.1.11 파이썬 베타 테스트 버전은 어떻게 구합니까?

알파와 베타 배포판은 <https://www.python.org/downloads/> 에서 확인할 수 있습니다. 모든 배포는 `comp.lang.python` 과 `comp.lang.python.announce` 뉴스그룹, 그리고 파이썬 홈페이지 <https://www.python.org/> 에서 공지됩니다. 뉴스의 RSS 피드가 제공됩니다.

Git을 통해 파이썬의 개발 버전을 사용할 수 있습니다. [파이썬 개발자 가이드](#)를 확인하십시오.

1.1.12 파이썬 버그 리포트나 패치는 어떻게 제출합니까?

To report a bug or submit a patch, use the issue tracker at <https://github.com/python/cpython/issues>.

파이썬이 어떻게 개발되는지 더 알고 싶으시다면 파이썬 개발자 가이드를 참조하십시오.

1.1.13 제가 참고할만한 파이썬 관련 기사가 있습니까?

당신이 가장 좋아하는 파이썬 책을 인용하는 것이 제일 좋을 것 같습니다.

The [very first article](#) about Python was written in 1991 and is now quite outdated.

Guido van Rossum and Jelke de Boer, “Interactively Testing Remote Servers Using the Python Programming Language”, CWI Quarterly, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283–303.

1.1.14 파이썬 관련 책이 있습니까?

예, 많이 있고 현재 출판되는 책들도 있습니다. 리스트를 원한다면 <https://wiki.python.org/moin/PythonBooks> 에서 파이썬 위키를 확인하십시오.

아니면 온라인 서점에서 “파이썬”을 검색하고 몬티 파이썬 관련 항목을 필터링해도 괜찮습니다; 검색할 때 “파이썬”과 “언어”를 검색해도 좋습니다.

1.1.15 도대체 www.python.org는 어디에 있는 겁니까?

The Python project’s infrastructure is located all over the world and is managed by the Python Infrastructure Team. Details [here](#).

1.1.16 왜 이 언어는 파이썬이라 불립니까?

Guido van Rossum 씨가 파이썬을 구현할 당시 그는 1970년대에 BBC에서 방영된 코미디 시리즈인 “몬티 파이톤의 플라잉 서커스”대본을 읽고 있었습니다. 판 로섬은 자신이 만든 언어가 짧고 독창적이면서도 약간 신비한 느낌의 이름이 필요하다고 생각했기에 언어를 파이썬이라고 부르기로 결정했습니다.

1.1.17 “몬티 파이톤의 플라잉 서커스”를 좋아해야합니까?

아니요, 하지만 도움은 됩니다. :)

1.2 현실 속의 파이썬

1.2.1 파이썬은 어느 정도 안정화 되어있습니까?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. As of version 3.9, Python will have a new feature release every 12 months ([PEP 602](#)).

The developers issue bugfix releases of older versions, so the stability of existing releases gradually improves. Bugfix releases, indicated by a third component of the version number (e.g. 3.5.3, 3.6.2), are managed for stability; only fixes for known problems are included in a bugfix release, and it’s guaranteed that interfaces will remain the same throughout a series of bugfix releases.

The latest stable releases can always be found on the [Python download page](#). Python 3.x is the recommended version and supported by most widely used libraries. Python 2.x **is not maintained anymore**.

1.2.2 얼마나 많은 사람이 파이썬을 씁니까?

아마 수백만의 사용자가 있을 텐데, 정확한 수를 얻기는 힘듭니다.

파이썬은 무료로 다운로드받을 수 있기 때문에 판매액이 없습니다. 그리고 파이썬은 다양한 사이트에서, 다양한 리눅스 배포판과 패키지로 제공되므로 다운로드 통계치 역시 모든 것을 알려주지 못합니다.

활발하게 활동하는 [comp.lang.python](#)이라는 뉴스그룹이 있지만 모든 파이썬 유저들이 그곳에 글을 올리지 않으며 심지어 읽는 것도 아닙니다.

1.2.3 파이썬으로 진행된 중요한 프로젝트가 존재합니까?

<https://www.python.org/about/success> 에서 파이썬을 사용한 프로젝트를 확인할 수 있습니다. 과거 파이썬 회의들을 통해 파이썬의 성장에 기여한 다양한 기업들과 단체들을 확인할 수 있습니다.

High-profile Python projects include the [Mailman mailing list manager](#) and the [Zope application server](#). Several Linux distributions, most notably [Red Hat](#), have written part or all of their installer and system administration software in Python. Companies that use Python internally include Google, Yahoo, and Lucasfilm Ltd.

1.2.4 파이썬에 어떤 새로운 개발요소들이 예정되어 있습니까?

See <https://peps.python.org/> for the Python Enhancement Proposals (PEPs). PEPs are design documents describing a suggested new feature for Python, providing a concise technical specification and a rationale. Look for a PEP titled “Python X.Y Release Schedule”, where X.Y is a version that hasn’t been publicly released yet.

New development is discussed on the [python-dev mailing list](#).

1.2.5 기존의 파이썬과 호환 가능하지 않은 변경안을 제시해도 괜찮습니까?

일반적으로, 아니요. 현재 전 세계에 수 없이 많은 파이썬 코드들이 작성되어 있으며, 이미 존재하는 프로그램의 아주 작은 부분이라도 무효화한다면 그 변경안은 눈살을 찌푸리고 봐야합니다. 당신이 변환 프로그램을 제공한다고 해도 설명서를 전부 업데이트 해야하는 문제가 남아있습니다. 뿐만 아니라 수 많은 책들이 파이썬에 관련해서 쓰여져 있는데, 저희는 한 순간에 그 모든 책들을 무효화 시키고 싶지 않습니다.

만약 기능이 바뀌어야 한다면 점차적인 업그레이드 방안의 제시를 필요로 합니다. [PEP 5](#)에서 하위호환 불가능한 변경을 도입할 때 사용자의 불편함을 최소화 하는 절차를 확인할 수 있습니다.

1.2.6 파이썬이 초보 프로그래머들에게 좋은 언어입니까?

예.

파스칼, C, 또는 C++나 Java의 일부처럼 절차적이고 정적인 언어를 학생들의 첫 언어로 가르치는 것은 아직 흔한 교육 방식입니다. 하지만 그들이 파이썬을 첫 언어로 배우는 것이 더 이로울 수도 있습니다. 파이썬은 굉장히 간단하면서도 일관된 문법과 방대한 표준 라이브러리를 가지고 있으며, 무엇보다 초기 프로그래밍 교육에서 파이썬을 이용하면 학생들이 자료형 디자인, 그리고 문제 분석과 같은 중요한 프로그래밍 능력에 집중할 수 있도록 해줍니다. 학생들은 파이썬을 통해 반복문과 절차 같은 기초적인 개념을 빠르게 접할 수 있습니다. 더 나아가 그들은 첫 강의에서 사용자 정의 객체를 활용하여 작업을 할 수도 있을 것입니다.

프로그래밍을 한 번도 해본 적이 없는 학생이라면 정적인 프로그래밍 언어를 사용하는 것은 자연스럽게 느껴지지 않을 것입니다. 그 과정을 해결하기 위해서는 부가적인 요소를 익혀야하며, 이는 강의 속도를 낮춥니다. 학생들은 컴퓨터처럼 사고하는 방식, 문제를 분석하는 방법, 일정한 인터페이스를 디자인하는 방법, 그리고 정보를 캡슐화하는 것을 배우려 합니다. 장기적으로 본다면 정적인 언어를 배우는 것이 중요할 수도 있지만, 학생들의 첫 프로그래밍 언어로 보았을 때는 좋다고 볼 수 없습니다.

파이썬이 첫 개발 언어로 좋은 이유로 다양한 요소들을 뽑습니다. 파이썬은 자바처럼 방대한 표준 라이브러리를 가지고 있어서 학생들이 수업의 극 초반에 뭔가 진짜 하는 프로그래밍 프로젝트를 경험할 수 있습니다. 네 개의 함수로 만들어보는 계산기나 가계부 프로그램이 과제의 전부가 아닙니다. 표준 라이브러리를 사용하므로써 학생들은 프로그래밍의 기본을 배우며 그럴싸한 애플리케이션을 만들어 볼 수 있습니다. 표준 라이브러리를 사용하면서 학생들에게 코드의 재활용 역시 가르쳐 줄 수 있습니다. 뿐만 아니라 PyGame 같은 제삼자 모듈 역시 학생들의 시야를 넓히는데 도움이 됩니다.

파이썬의 상호작용 가능한 인터프리터는 학생들이 코딩하며 언어의 기능을 테스트해 볼 수 있게 해줍니다. 학생들은 하나의 창에서 코드를 수정하며 다른 창에서 인터프리터를 돌려볼 수 있습니다. 만약 그들이 리스트에 대한 메서드가 기억나지 않는다면, 이와 같은 시도를 해볼 수 있습니다:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
↪ 'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

인터프리터가 있다면, 학생들이 코딩하면서 설명서를 열람하기 쉽습니다.

There are also good IDEs for Python. IDLE is a cross-platform IDE for Python that is written in Python using Tkinter. Emacs users will be happy to know that there is a very good Python mode for Emacs. All of these programming environments provide syntax highlighting, auto-indenting, and access to the interactive interpreter while coding. Consult [the Python wiki](#) for a full list of Python editing environments.

만약 파이썬이 교육 분야에서 어떻게 사용되는지에 대해 논의하고 싶으시면 [the edu-sig mailing list](#)에 들어와 주십시오.

2.1 일반적인 질문

2.1.1 중단점, 단일 스텝핑(single-stepping) 등을 포함하는 소스 코드 수준 디버거가 있습니까?

예.

파이썬을 위한 여러 디버거가 아래에 설명되어 있으며, 내장 함수 `breakpoint()` 를 사용하면 이들 중 하나로 들어갈 수 있습니다.

`pdb` 모듈은 간단하지만 적절한 파이썬 용 콘솔 모드 디버거입니다. 표준 파이썬 라이브러리의 일부이며, [파이썬 3.7 릴리스 매뉴얼](#)에서 설명하고 있습니다. `pdb`의 코드를 예로 사용하여 자체 디버거를 작성할 수도 있습니다.

The IDLE interactive development environment, which is part of the standard Python distribution (normally available as [Tools/scripts/idle3](#)), includes a graphical debugger.

PythonWin is a Python IDE that includes a GUI debugger based on `pdb`. The PythonWin debugger colors breakpoints and has quite a few cool features such as debugging non-PythonWin programs. PythonWin is available as part of [pywin32](#) project and as a part of the [ActivePython](#) distribution.

[Eric](#) is an IDE built on PyQt and the Scintilla editing component.

[trepan3k](#) is a gdb-like debugger.

[Visual Studio Code](#) is an IDE with debugging tools that integrates with version-control software.

그래픽 디버거를 포함하는 많은 상용 파이썬 IDE가 있습니다. 다음을 포함합니다:

- [Wing IDE](#)
- [Komodo IDE](#)
- [PyCharm](#)

2.1.2 버그를 찾거나 정적 분석을 수행하는 데 도움이 되는 도구가 있습니까?

예.

[Pylint](#) and [Pyflakes](#) do basic checking that will help you catch bugs sooner.

Static type checkers such as [Mypy](#), [Pyre](#), and [Pytype](#) can check type hints in Python source code.

2.1.3 파이썬 스크립트로 독립 실행형 바이너리를 만들려면 어떻게 해야 하나요?

사용자가 파이썬 배포를 먼저 설치하지 않고도 다운로드하여 실행할 수 있는 독립 실행형 프로그램을 원하는 것이 전부라면 파이썬을 C 코드로 컴파일하는 기능이 필요하지는 않습니다. 프로그램에 필요한 모듈 집합을 파악하고 이러한 모듈들을 파이썬 바이너리와 결합하여 단일 실행 파일을 생성하는 많은 도구가 있습니다.

One is to use the freeze tool, which is included in the Python source tree as `Tools/freeze`. It converts Python byte code to C arrays; with a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

소스를 재귀적으로 검색하여 `import` 문(두 형식 모두)을 찾고 표준 파이썬 경로뿐만 아니라 소스 디렉터리에서 모듈을 찾습니다 (내장할 모듈을 위해). 그런 다음 파이썬으로 작성된 모듈의 바이트 코드를 C 코드 (marshal 모듈을 사용하여 코드 객체로 변환할 수 있는 배열 초기화기)로 바꾸고 프로그램에서 실제로 사용되는 내장 모듈만 포함하는 특별한 구성 파일을 만듭니다. 그런 다음 생성된 C 코드를 컴파일하고 이를 나머지 파이썬 인터프리터와 링크하여 스크립트와 똑같이 작동하는 자체 포함 바이너리를 형성합니다.

The following packages can help with the creation of console and GUI executables:

- `Nuitka` (Cross-platform)
- `PyInstaller` (Cross-platform)
- `PyOxidizer` (Cross-platform)
- `cx_Freeze` (Cross-platform)
- `py2app` (macOS only)
- `py2exe` (Windows only)

2.1.4 파이썬 프로그램을 위한 코딩 표준이나 스타일 지침서가 있습니까?

예. 표준 라이브러리 모듈에 요구되는 코딩 스타일은 **PEP 8**에서 설명합니다.

2.2 핵심 언어

2.2.1 변수에 값이 있을 때 `UnboundLocalError` 가 발생하는 이유는 무엇입니까?

It can be a surprise to get the `UnboundLocalError` in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

이 코드는:

```
>>> x = 10
>>> def bar():
...     print(x)
...
>>> bar()
10
```

작동하지만, 이 코드는:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

results in an `UnboundLocalError`:

```
>>> foo()
Traceback (most recent call last):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
UnboundLocalError: local variable 'x' referenced before assignment
```

이는 스코프에서 변수에 대입할 때, 해당 변수가 그 스코프에 대해 지역(local)이 되고 외부 스코프에서 비슷한 이름의 변수를 가리키기 때문입니다. foo의 마지막 문장은 x에 새 값을 대입하므로, 컴파일러는 이 값을 지역 변수로 인식합니다. 결과적으로 앞의 print(x)가 초기화되지 않은 지역 변수를 인쇄하려고 할 때 예외가 발생합니다.

위의 예에서 변수를 전역(global)으로 선언하여 외부 스코프 변수에 액세스 할 수 있습니다:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
...
>>> foobar()
10
```

이 명시적 선언은 (클래스와 인스턴스 변수의 표면적으로 유사한 상황과 달리) 실제로 외부 스코프에 있는 변수의 값을 수정하고 있음을 상기시키기 위해 필요합니다:

```
>>> print(x)
11
```

nonlocal 키워드를 사용하여 중첩된 스코프에서 비슷한 일을 할 수 있습니다:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
...
>>> foo()
10
11
```

2.2.2 파이썬에서 지역과 전역 변수에 대한 규칙은 무엇입니까?

파이썬에서, 함수 내에서 참조되기만 하는 변수는 묵시적으로 전역입니다. 변수가 함수 본문 내 어디에서든 값을 대입하면, 명시적으로 전역으로 선언되지 않는 한 지역으로 간주합니다.

처음에는 조금 의외지만, 잠시 생각해 보면 이해가 됩니다. 한편으로, 대입된 변수에 global을 요구하면 의도하지 않은 부작용에 대한 저지선을 제공합니다. 반면에, 모든 전역 참조에 global이 요구된다면, 항상 global을 사용하게 됩니다. 내장 함수나 импорт 한 모듈의 구성 요소에 대한 모든 참조를 전역으로 선언해야 합니다. 이 혼란은 부작용을 식별하기 위한 global 선언의 유용성을 무효로 합니다.

2.2.3 다른 값으로 루프에서 정의된 람다는 왜 모두 같은 결과를 반환합니까?

for 루프를 사용하여 몇 가지 다른 람다(또는 일반 함수조차)를 정의한다고 가정하십시오, 예를 들어:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

이것은 $x**2$ 를 계산하는 5개의 람다가 포함된 리스트를 제공합니다. 호출되면, 각각 0, 1, 4, 9 및 16을 반환할 것으로 예상할 수 있습니다. 그러나, 실제로 시도하면 모두 16을 반환한다는 것을 알 수 있습니다:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

이는 x 가 람다에 지역(local)이 아니라 외부 스코프에 정의되어 있기 때문에 발생하며, 람다가 호출될 때 액세스됩니다 — 정의될 때가 아닙니다. 루프의 끝에서, x 의 값은 4이므로, 모든 함수는 이제 $4**2$, 즉 16을 반환합니다. x 의 값을 변경하고 람다의 결과가 어떻게 변경되는지 봄으로써 이를 확인할 수도 있습니다:

```
>>> x = 8
>>> squares[2]()
64
```

이를 피하려면, 람다에 대해 지역인 변수에 값을 저장하여, 전역 x 의 값에 의존하지 않도록 할 필요가 있습니다:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

여기서 $n=x$ 는 람다에 지역인 새 변수 n 을 만들고, 루프의 해당 시점에서 x 와 같은 값을 갖도록 람다가 정의될 때 계산됩니다. 이는 n 의 값이 첫 번째 람다에서 0, 두 번째에서 1, 세 번째에서 2 등이 됨을 의미합니다. 따라서 각 람다는 이제 올바른 결과를 반환합니다:

```
>>> squares[2]()
4
>>> squares[4]()
16
```

이 동작은 람다에만 국한된 것이 아니라 일반 함수에도 적용됩니다.

2.2.4 모듈 간에 전역 변수를 공유하려면 어떻게 해야 합니까?

단일 프로그램 내에서 모듈 간에 정보를 공유하는 규범적인 방법은 특별한 모듈(종종 `config`나 `cfg`라고 불립니다)을 만드는 것입니다. 응용 프로그램의 모든 모듈에서 `config` 모듈을 임포트 하기만 하면 됩니다; 그러면 모듈이 전역 이름으로 사용 가능해집니다. 각 모듈의 인스턴스는 오직 하나이기 때문에, 모듈 객체에 대한 변경 사항은 모든 곳에 반영됩니다. 예를 들면 다음과 같습니다:

`config.py`:

```
x = 0    # Default value of the 'x' configuration setting
```

`mod.py`:

```
import config
config.x = 1
```

`main.py`:

```
import config
import mod
print(config.x)
```

Note that using a module is also the basis for implementing the singleton design pattern, for the same reason.

2.2.5 모듈에서 임포트를 사용하는 “모범 사례”는 무엇입니까?

일반적으로, `from modulename import *`를 사용하지 마십시오. 그렇게 하면 임포트 하는 곳의 이름 공간이 어수선해지고, 린터(linter)가 정의되지 않은 이름을 감지하기가 훨씬 어려워집니다.

파일 맨 위에서 모듈을 임포트 하십시오. 그렇게 하면 코드에 필요한 다른 모듈을 명확하게 하고 모듈 이름이 스코프에 있는지에 대한 질문을 피할 수 있습니다. 한 줄에 하나의 임포트를 사용하면 모듈 임포트를 쉽게 추가하고 삭제할 수 있지만, 한 줄에 여러 임포트를 사용하면 화면 공간을 덜 사용합니다.

다음 순서로 모듈을 임포트 하는 것이 좋습니다:

1. standard library modules – e.g. `sys`, `os`, `argparse`, `re`
2. third-party library modules (anything installed in Python’s site-packages directory) – e.g. `dateutil`, `requests`, `PIL.Image`
3. locally developed modules

순환 임포트 관련 문제를 피하고자 임포트를 함수나 클래스로 이동해야 하는 경우가 있습니다. Gordon McMillan은 다음과 같이 말했습니다:

두 모듈 모두 “`import <module>`” 형식의 임포트를 사용하면 순환 임포트는 괜찮습니다. 두 번째 모듈이 첫 번째 모듈의 이름(name)을 붙잡으려고 하고 (“`from module import name`”) 임포트가 최상위 수준에 있으면 실패합니다. 첫 번째 모듈이 두 번째 모듈을 임포트 하는 중이라서 첫 번째 모듈에 있는 이름을 아직 사용할 수 없기 때문입니다.

이 경우, 두 번째 모듈이 하나의 함수에서만 사용된다면, 임포트를 해당 함수로 쉽게 이동할 수 있습니다. 임포트가 호출될 때, 첫 번째 모듈의 초기화가 완료되었고, 두 번째 모듈은 임포트를 수행할 수 있습니다.

일부 모듈이 플랫폼 특정이면 임포트를 코드의 최상위 수준에서 다른 곳으로 이동해야 할 수도 있습니다. 이 경우, 파일 맨 위에서 모든 모듈을 임포트 하는 것이 가능하지 않을 수도 있습니다. 이 경우, 해당 플랫폼 특정 코드에서 올바른 모듈을 임포트 하는 것이 좋은 선택입니다.

순환 임포트를 피하거나 모듈의 초기화 시간을 줄이려는 등의 문제를 해결하는 데 필요할 때만, 함수 정의 내부와 같은 지역 스코프로 임포트를 옮기십시오. 이 기법은 프로그램 실행 방법에 따라 많은 임포트가 필요하지 않을 때 특히 유용합니다. 모듈이 해당 함수에서만 사용될 때 임포트를 함수로 옮기고 싶을 수도 있습니다. 모듈의 일회성 초기화 때문에 모듈을 처음 로드하는 데 비용이 많이 들 수 있지만, 모듈을 여러 번 로드하는 것은 사실상 무료임에 유의하십시오, 두 번의 디서너리 조회만 발생합니다. 모듈 이름이 스코프를 벗어난 경우에도, 모듈은 아마도 `sys.modules`에 있을 겁니다.

2.2.6 객체 간에 기본값이 공유되는 이유는 무엇입니까?

이 유형의 버그는 흔히 신참 프로그래머들을 깨웁니다. 이 함수를 생각해보십시오:

```
def foo(mydict={}): # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

이 함수를 처음 호출하면, `mydict`에 단일 항목이 포함됩니다. 두 번째는, `foo()`가 실행되기 시작할 때, `mydict`가 이미 항목에 들어 있는 상태로 시작하기 때문에, `mydict`가 두 개의 항목을 포함합니다.

종종 함수 호출이 기본값으로 새 객체를 만들 것으로 기대합니다. 그렇게 되지 않습니다. 함수가 정의될 때, 기본값은 정확히 한 번 만들어집니다. 이 예제의 디서너리와 같이, 해당 객체가 변경되면, 함수에 대한 후속 호출은 이 변경된 객체를 참조합니다.

정의에 따라, 숫자, 문자열, 튜플 및 `None`과 같은 불변 객체는 변경에 안전합니다. 디서너리, 리스트 및 클래스 인스턴스와 같은 가변 객체를 변경하면 혼란스러울 수 있습니다.

이 기능으로 인해, 가변 객체를 기본값으로 사용하지 않는 것이 좋습니다. 대신, `None`을 기본값으로 사용하고 함수 내부에서 매개변수가 `None`인지 확인한 다음 새 리스트/디서너리/무엇이든 새로 만드십시오. 예를 들어, 다음과 같이 쓰지 마십시오:

```
def foo(mydict={}):
    ...
```


대신 이렇게 쓰십시오:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {} # create a new dict for local namespace
```

이 기능은 유용할 수 있습니다. 계산하는 데 시간이 걸리는 함수가 있을 때, 일반적인 기법은 각 함수 호출의 매개변수와 결과값을 캐시하고, 같은 값이 다시 요청되면 캐시된 값을 반환하는 것입니다. 이것을 “memoizing” 이라고 하며, 다음과 같이 구현할 수 있습니다:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result # Store result in the cache
    return result
```

기본값 대신 딕셔너리를 포함하는 전역 변수를 사용할 수 있습니다; 취향의 문제입니다.

2.2.7 한 함수에서 다른 함수로 선택적이나 키워드 매개변수를 전달하려면 어떻게 해야 합니까?

함수의 매개변수 목록에 *와 ** 지정자를 사용하여 인자를 수집하십시오; 이것은 위치 인자를 튜플로, 키워드 인자를 딕셔너리로 제공합니다. 그런 다음 *와 **를 사용하여 다른 함수를 호출할 때 이러한 인자를 전달할 수 있습니다:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

2.2.8 인자와 매개변수의 차이점은 무엇입니까?

Parameters are defined by the names that appear in a function definition, whereas *arguments* are the values actually passed to a function when calling it. Parameters define what *kind of arguments* a function can accept. For example, given the function definition:

```
def func(foo, bar=None, **kwargs):
    pass
```

foo, *bar* 및 *kwargs*는 *func*의 매개변수입니다. 그러나, *func*를 호출할 때, 예를 들면:

```
func(42, bar=314, extra=somevar)
```

42, 314 및 *somevar* 값은 인자입니다.

2.2.9 리스트 ‘y’를 변경할 때 리스트 ‘x’도 변경되는 이유는 무엇입니까?

다음과 같은 코드를 작성하면:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
[10]
>>> x
[10]
```

`y`에 요소를 추가하면 `x`도 변경되는 이유가 궁금할 것입니다.

이 결과를 만드는 두 가지 요소가 있습니다:

- 1) 변수는 단순히 객체를 가리키는 이름입니다. `y = x`를 수행하면 리스트의 사본을 만들지 않습니다 - `x`가 참조하는 것과 같은 객체를 참조하는 새 변수 `y`를 만듭니다. 이는 하나의 객체(리스트)만 있고, `x`와 `y` 모두 그 객체를 참조함을 의미합니다.
- 2) 리스트는 가변입니다, 내용을 변경할 수 있다는 뜻입니다.

After the call to `append()`, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

대신 불변 객체를 `x`에 대입하면:

```
>>> x = 5 # ints are immutable
>>> y = x
>>> x = x + 1 # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

이 경우 `x`와 `y`가 더는 같지 않다는 것을 알 수 있습니다. 이는 정수가 불변이기 때문이고, `x = x + 1`을 수행할 때 값을 증가시켜서 정수 5를 변경하는 것이 아닙니다; 대신 새 객체(정수 6)를 만들어 `x`에 대입합니다 (즉, `x`가 참조하는 객체를 바꿉니다). 이 대입 후에는 두 개의 객체(정수 6과 5)와 이를 참조하는 두 개의 변수를 갖게 됩니다 (`x`는 이제 6을 참조하지만, `y`는 여전히 5를 참조합니다).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and `sorted(y)`) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily diagnosed error.

그러나, 같은 연산이 때때로 형에 따라 다른 동작을 갖는 한 가지 연산 클래스가 있습니다: 증분 대입 연산자. 예를 들어, `+=`는 리스트를 변경하지만, 튜플이나 정수는 변경하지 않습니다 (`a_list += [1, 2, 3]`은 `a_list.extend([1, 2, 3])`과 동등하고 `a_list`를 변경하지만, `some_tuple += (1, 2, 3)`과 `some_int += 1`은 새 객체를 만듭니다).

달리 표현하면:

- 가변 객체(`list`, `dict`, `set` 등)가 있으면, 일부 특정 연산을 사용하여 객체를 변경하면 해당 객체를 참조하는 모든 변수가 변경을 보게 됩니다.
- 불변 객체(`str`, `int`, `tuple` 등)가 있으면, 이를 참조하는 모든 변수는 항상 같은 값을 보게 되지만, 해당 값을 새로운 값으로 변환하는 연산은 항상 새로운 객체를 반환합니다.

두 변수가 같은 객체를 참조하는지를 알고 싶다면, `is` 연산자나 내장 함수 `id()`를 사용할 수 있습니다.

2.2.10 출력 매개변수가 있는 함수를 작성하려면 어떻게 해야 합니까 (참조에 의한 호출)?

파이썬에서 인자는 대입으로 전달됨을 기억하십시오. 대입은 단지 객체에 대한 참조를 만들기 때문에, 호출자와 피호출자의 인자 이름 간에 에일리어스가 없고, 참조에 의한 호출도 없습니다. 여러 가지 방법으로 원하는 효과를 얻을 수 있습니다.

- 1) 결과의 튜플을 반환하여:

```
>>> def func1(a, b):
...     a = 'new-value'           # a and b are local names
...     b = b + 1                 # assigned to new objects
...     return a, b              # return new values
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

이것은 거의 항상 가장 명확한 해법입니다.

- 2) 전역 변수를 사용하여. 이것은 스레드 안전하지 않고, 권장하지 않습니다.
- 3) 가변 (제자리에서 변경할 수 있는) 객체를 전달하여:

```
>>> def func2(a):
...     a[0] = 'new-value'        # 'a' references a mutable list
...     a[1] = a[1] + 1           # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

- 4) 변경되는 딕셔너리를 전달하여:

```
>>> def func3(args):
...     args['a'] = 'new-value'    # args is a mutable dictionary
...     args['b'] = args['b'] + 1  # change it in-place
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
>>> args
{'a': 'new-value', 'b': 100}
```

- 5) 또는 클래스 인스턴스에 값을 묶어서:

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'       # args is a mutable Namespace
...     args.b = args.b + 1        # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

이렇게 복잡하게 만들어야 할 좋은 이유는 거의 없습니다.

최선의 선택은 여러 결과가 포함된 튜플을 반환하는 것입니다.

2.2.11 파이썬에서 고차 함수(higher order function)를 어떻게 만드나요?

두 가지 선택이 있습니다: 중첩된 스코프를 사용하거나 콜러블 객체를 사용할 수 있습니다. 예를 들어, 값 $a \cdot x + b$ 를 계산하는 함수 $f(x)$ 를 반환하는 `linear(a,b)`를 정의하려고 한다고 가정하십시오. 중첩된 스코프를 사용해서:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

또는 콜러블 객체를 사용해서:

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

두 경우 모두,

```
taxes = linear(0.3, 2)
```

`taxes(10e6) == 0.3 * 10e6 + 2`가 되도록 하는 콜러블 객체를 제공합니다.

콜러블 객체 접근 방식은 약간 느리고 코드가 약간 길어진다는 단점이 있습니다. 그러나, 콜러블 컬렉션은 상속을 통해 서명을 공유할 수 있습니다:

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

객체는 여러 메서드의 상태를 캡슐화 할 수 있습니다:

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

여기서 `inc()`, `dec()` 및 `reset()`은 같은 계수 변수를 공유하는 함수처럼 작동합니다.

2.2.12 파이썬에서 객체를 어떻게 복사합니까?

일반적으로, 일반적인 때 `copy.copy()`나 `copy.deepcopy()`를 시도하십시오. 모든 객체를 복사할 수는 없지만, 대부분 가능합니다.

일부 객체는 더 쉽게 복사할 수 있습니다. 딕셔너리에는 `copy()` 메서드가 있습니다:

```
newdict = olddict.copy()
```

시퀀스는 슬라이싱으로 복사할 수 있습니다:

```
new_l = l[:]
```

2.2.13 객체의 메서드나 어트리뷰트를 어떻게 찾을 수 있습니까?

For an instance `x` of a user-defined class, `dir(x)` returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

2.2.14 코드에서 객체 이름을 어떻게 찾을 수 있습니까?

일반적으로 말하자면, 객체에는 실제로 이름이 없기 때문에 그럴 수 없습니다. 기본적으로, 대입은 항상 이름을 값에 연결합니다; `def`와 `class` 문의 경우도 마찬가지이지만, 이 경우 값은 콜러블입니다. 다음 코드를 고려하십시오:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two names and invoked through the name `B` the created instance is still reported as an instance of class `A`. However, it is impossible to say whether the instance's name is `a` or `b`, since both names are bound to the same value.

일반적으로 말해 코드가 특정 값의 “이름을 알아야” 할 필요는 없습니다. 의도적으로 내성적인(introspective) 프로그램을 작성하지 않는 한, 이는 일반적으로 접근 방식의 변경이 도움이 될 수 있다는 신호입니다.

`comp.lang.python`에서, Fredrik Lundh는 언젠가 이 질문에 대해 훌륭한 비유를 했습니다:

여러분이 현관에서 발견한 고양이의 이름을 얻는 것과 같은 방법: 고양이(객체) 자체는 여러분에게 자신의 이름을 말할 수 없고, 전혀 신경 쓰지도 않습니다 - 따라서 그것이 어떻게 불리는지 알아내는 유일한 방법은 여러분 이웃 모두(이름 공간)에게 자신의 고양이(객체)인지 묻는 것입니다...

... 여러 이름으로 알려져 있거나 전혀 이름이 없다는 것을 알게 되더라도 놀라지 마십시오!

2.2.15 쉼표 연산자의 우선순위는 어떻게 되나요?

쉼표는 파이썬에서 연산자가 아닙니다. 이 세션을 고려하십시오:

```
>>> "a" in "b", "a"
(False, 'a')
```

쉼표는 연산자가 아니라 표현식 사이의 구분자이기 때문에 위는 다음과 같이 입력한 것처럼 평가됩니다:

```
("a" in "b"), "a"
```

다음과 같이 평가되지 않습니다:

```
"a" in ("b", "a")
```

다양한 대입 연산자(`=`, `+=` 등)도 마찬가지입니다. 이들은 실제로 연산자가 아니라 대입 문의 문법 구분자입니다.

2.2.16 C의 “?:” 삼항 연산자와 동등한 것이 있습니까?

예, 있습니다. 문법은 다음과 같습니다:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

이 문법이 파이썬 2.5에서 소개되기 전에, 일반적인 관용구는 논리 연산자를 사용하는 것이었습니다:

```
[expression] and [on_true] or [on_false]
```

그러나, 이 관용구는 안전하지 않습니다. `on_true`가 거짓 불리언 값을 가질 때 잘못된 결과가 나올 수 있습니다. 따라서, 항상 ... `if` ... `else` ... 형식을 사용하는 것이 좋습니다.

2.2.17 파이썬에서 난독화된 한 줄 코드를 작성할 수 있습니까?

Yes. Usually this is done by nesting `lambda` within `lambda`. See the following three examples, slightly adapted from Ulf Bartelt:

```
from functools import reduce

# Primes < 1000
print(list(filter(None, map(lambda y: y * reduce(lambda x, y: x * y != 0,
map(lambda x, y: y % x, range(2, int(pow(y, 0.5) + 1))), 1), range(2, 1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x, f: lambda x, f: (f(x-1, f) + f(x-2, f)) if x > 1 else 1:
f(x, f), range(10))))

# Mandelbrot set
print((lambda Ru, Ro, Iu, Io, IM, Sx, Sy: reduce(lambda x, y: x + '\n' + y, map(lambda y,
Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, Sy=Sy, L=lambda yc, Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, i=IM,
Sx=Sx, Sy=Sy: reduce(lambda x, y: x + y, map(lambda x, xc=Ru, yc=yc, Ru=Ru, Ro=Ro,
i=i, Sx=Sx, F=lambda xc, yc, x, y, k, f: lambda xc, yc, x, y, k, f: (k <= 0) or (x * x + y * y
>= 4.0) or 1 + f(xc, yc, x * x - y * y + xc, 2.0 * x * y + yc, k - 1, f): f(xc, yc, x, y, k, f): chr(
64 + F(Ru + x * (Ro - Ru) / Sx, yc, 0, 0, i)), range(Sx)): L(Iu + y * (Io - Iu) / Sy), range(Sy
)))) (-2.1, 0.7, -1.2, 1.2, 30, 80, 24))

#      \__ _/  \__ _/  /   /   /__ lines on screen
#          V          V   /   /__ columns on screen
#          /           /__ maximum of "iterations"
#          /           /__ range on y axis
#          /           /__ range on x axis
```

집에서 이것을 시도하지 마십시오, 어린이들!

2.2.18 함수의 매개변수 목록에서 슬래시(/)는 무엇을 의미합니까?

A slash in the argument list of a function denotes that the parameters prior to it are positional-only. Positional-only parameters are the ones without an externally usable name. Upon calling a function that accepts positional-only parameters, arguments are mapped to parameters based solely on their position. For example, `divmod()` is a function that accepts positional-only parameters. Its documentation looks like this:

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y). Invariant: div*y + mod == x.
```

매개변수 목록 끝의 슬래시는 두 매개변수가 위치 전용임을 의미합니다. 따라서, 키워드 인자로 `divmod()` 를 호출하면 에러가 발생합니다:

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

2.3 숫자와 문자열

2.3.1 16진수와 8진수 정수는 어떻게 지정합니까?

8진수를 지정하려면, 8진수 값 앞에 0을 붙이고, 소문자나 대문자 “o”를 붙입니다. 예를 들어, 변수 “a”를 8진수 값 “10” (10진수 8)으로 설정하려면, 이렇게 입력하십시오:

```
>>> a = 0o10
>>> a
8
```

16진수도 쉽습니다. 16진수 앞에 0을 붙이고, 소문자나 대문자 “x”를 붙이기만 하면 됩니다. 16진 숫자는 소문자나 대문자로 지정할 수 있습니다. 예를 들어, 파이썬 인터프리터에서:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

2.3.2 왜 `-22 // 10` 이 `-3`을 반환합니까?

주로 `i % j`가 `j`와 같은 부호를 갖도록 하려는 것입니다. 여러분이 이것을 원하고, 다음과 같은 것도 원한다면:

```
i == (i // j) * j + (i % j)
```

정수 나누기는 floor를 반환해야 합니다. C 또한 이 항등식을 만족하도록 요구하고, `i // j`를 자르는 (truncate) 컴파일러는 `i % j`가 `i`와 같은 부호를 갖도록 할 필요가 있습니다.

`j`가 음수인 경우 `i % j`에 대한 실제 사용 사례는 거의 없습니다. `j`가 양수이면, 많은 사례가 있으며, 사실상 모든 경우에 `i % j`가 ≥ 0 인 것이 더 유용합니다. 시계가 지금 10을 가리킨다면, 200시간 전에는 어디를 가리키겠습니까? $-190 \% 12 == 2$ 가 유용합니다; $-190 \% 12 == -10$ 은 물기를 기다리는 버그입니다.

2.3.3 How do I get int literal attribute instead of SyntaxError?

Trying to lookup an `int` literal attribute in the normal manner gives a `SyntaxError` because the period is seen as a decimal point:

```
>>> 1.__class__
File "<stdin>", line 1
  1.__class__
    ^
SyntaxError: invalid decimal literal
```

The solution is to separate the literal from the period with either a space or parentheses.

```
>>> 1.__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

2.3.4 문자열을 숫자로 어떻게 변환합니까?

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to a floating-point number, e.g. `float('144') == 144.0`.

기본적으로 이것은 숫자를 십진수로 해석합니다. 그래서 `int('0144') == 144`는 참이고 `int('0x144')`는 `ValueError`를 발생시킵니다. `int(string, base)`는 두 번째 선택적 인자로 변환에 사용할 진수(base)를 받아들이며, `int('0x144', 16) == 324`입니다. base가 0으로 지정되면, 숫자는 파이썬의 규칙을 사용하여 해석됩니다: 선행 '0o'는 8진수를 나타내고, '0x'는 16진수를 나타냅니다.

필요한 것이 문자열을 숫자로 변환하는 것뿐이라면 내장 함수 `eval()`을 사용하지 마십시오. `eval()`은 상당히 느리며 보안 위험을 초래할 수 있습니다: 누군가 원하지 않는 부작용이 있는 파이썬 표현식을 전달할 수 있습니다. 예를 들어, 누군가 여러분의 홈 디렉터리를 지우는 `__import__('os').system("rm -rf $HOME")`을 전달할 수 있습니다.

`eval()`은 또한 숫자를 파이썬 표현식으로 해석하는 효과가 있어서, 예를 들어 `eval('09')`는 파이썬이 ('0'을 제외한) 십진수에서 선행 '0'을 허용하지 않기 때문에 구문 에러가 발생합니다.

2.3.5 숫자를 문자열로 어떻게 변환합니까?

To convert, e.g., the number 144 to the string '144', use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the f-strings and formatstrings sections, e.g. `"{:04d}".format(144)` yields '0144' and `"{: .3f}".format(1.0/3.0)` yields '0.333'.

2.3.6 제자리에서 문자열을 어떻게 수정합니까?

그럴 수 없습니다. 문자열은 불변이기 때문입니다. 대부분의 경우, 조립하려는 다양한 부분으로 새 문자열을 구성해야 합니다. 그러나 제자리에서 유니코드 데이터를 수정할 수 있는 객체가 필요하다면, `io.StringIO` 객체나 `array` 모듈을 사용해보십시오:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('w', s)
>>> print(a)
array('w', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('w', 'yello, world')
>>> a.tounicode()
'yello, world'
```

2.3.7 문자열을 사용하여 어떻게 함수/메서드를 호출합니까?

다양한 기법이 있습니다.

- 문자열을 함수로 매핑하는 딕셔너리를 사용하는 것이 가장 좋습니다. 이 기법의 주요 장점은 문자열이 함수 이름과 일치할 필요가 없다는 것입니다. 이것은 또한 case 구문을 흉내 내는 데 사용되는 기본 기법입니다:

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b} # Note lack of parens for funcs

dispatch[get_input()]() # Note trailing parens to call function
```

- 내장 함수 `getattr()` 을 사용하십시오:

```
import foo
getattr(foo, 'bar')()
```

`getattr()` 은 클래스, 클래스 인스턴스, 모듈 등을 포함하는 모든 객체에서 작동함에 유의하십시오. 이것은 다음과 같이 표준 라이브러리의 여러 곳에서 사용됩니다:

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- 함수 이름을 해석(resolve)하려면 `locals()` 를 사용하십시오:

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()
```

2.3.8 Is there an equivalent to Perl's `chomp()` for removing trailing newlines from strings?

`s.rstrip("\r\n")` 을 사용하면 다른 후행 공백을 제거하지 않고 문자열 `s` 의 끝에 있는 모든 줄 종결자를 제거할 수 있습니다. 문자열 `s` 가 끝에 빈 줄이 여러 개 붙어 한 줄 이상을 나타내면, 모든 빈 줄의 줄 종결자가 제거됩니다:

```
>>> lines = ("line 1 \r\n"
...         "\r\n"
...         "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```


일반적으로 한 번에 한 줄씩 텍스트를 읽을 때만 필요하기 때문에, `s.rstrip()` 을 이런 식으로 사용하면 잘 작동합니다.

2.3.9 Is there a `scanf()` or `sscanf()` equivalent?

그런 식으로는 없습니다.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional “sep” parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C’s `sscanf` and better suited for the task.

2.3.10 What does `UnicodeDecodeError` or `UnicodeEncodeError` error mean?

unicode-howto를 참조하십시오.

2.3.11 Can I end a raw string with an odd number of backslashes?

A raw string ending with an odd number of backslashes will escape the string’s quote:

```
>>> r'C:\this\will\not\work\'
File "<stdin>", line 1
    r'C:\this\will\not\work\'
    ^
SyntaxError: unterminated string literal (detected at line 1)
```

There are several workarounds for this. One is to use regular strings and double the backslashes:

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

Another is to concatenate a regular string containing an escaped backslash to the raw string:

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

It is also possible to use `os.path.join()` to append a backslash on Windows:

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

Note that while a backslash will “escape” a quote for the purposes of determining where the raw string ends, no escaping occurs when interpreting the value of the raw string. That is, the backslash remains present in the value of the raw string:

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

Also see the specification in the language reference.

2.4 성능

2.4.1 내 프로그램이 너무 느립니다. 속도를 높이려면 어떻게 해야 하나요?

그것은 일반적으로 힘든 일입니다. 먼저, 더 뛰어들기 전에 기억해야 할 사항이 있습니다:

- 성능 특성은 파이썬 구현마다 다릅니다. 이 FAQ는 *CPython*에 중점을 둡니다.

- 동작은 운영 체제마다 다를 수 있습니다, 특히 I/O 나 다중 스레드에 관해 이야기할 때 그렇습니다.
- 코드를 최적화하려고 시도하기 전에 프로그램에서 항상 핫스팟을 찾아야 합니다 (profile 모듈을 참조하십시오).
- 벤치마크 스크립트를 작성하면 개선 사항을 탐색할 때 빠르게 반복할 수 있습니다 (timeit 모듈 참조).
- 정교한 최적화에 숨겨진 회귀(regressions)를 잠재적으로 도입하기 전에 (단위 테스트나 기타 기법을 통해) 우수한 코드 커버리지를 갖는 것이 좋습니다.

이것을 전제로, 파이썬 코드 속도를 높이는 많은 트릭이 있습니다. 다음은 수용 가능한 성능 수준에 도달하기 위해 먼 길을 갈 때 도움이 되는 몇 가지 일반적인 원칙입니다:

- 알고리즘을 더 빠르게 만들면 (또는 더 빠른 알고리즘으로 변경하면) 코드 전체에 미세 최적화 트릭을 뿌리는 것보다 훨씬 큰 이점을 얻을 수 있습니다.
- 올바른 데이터 구조를 사용하십시오. `builtin-types`과 `collections` 모듈에 대한 설명서를 연구하십시오.
- 표준 라이브러리가 무언가를 하기 위한 프리미티브를 제공할 때, 여러분이 떠올린 다른 대안보다 빠를 가능성이 높습니다 (보장되지는 않습니다). 이것은 내장과 일부 확장형과 같이 C로 작성된 프리미티브의 경우에는 두 배로 그렇습니다. 예를 들어, 정렬하려면 `list.sort()` 내장 메서드나 관련 `sorted()` 함수를 사용하십시오 (그리고 약간 고급 사용법의 예는 `sortingshowto`를 참조하십시오).
- 추상화는 간접(indirections)을 만드는 경향이 있고 인터프리터가 더 많은 일을 하도록 강요합니다. 간접의 수준이 유용한 작업의 양을 초과하면, 프로그램 속도가 느려집니다. 과도한 추상화를 피해야 합니다, 특히 작은 함수나 메서드의 형태에서 그렇습니다 (종종 가독성에도 해롭습니다).

If you have reached the limit of what pure Python can allow, there are tools to take you further away. For example, [Cython](#) can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms. Cython can take advantage of compilation (and optional type annotations) to make your code significantly faster than when interpreted. If you are confident in your C programming skills, you can also write a C extension module yourself.

➡ 더 보기

성능 팁에 할당된 위키 페이지.

2.4.2 많은 문자열을 함께 이어붙이는 가장 효율적인 방법은 무엇입니까?

`str`과 `bytes` 객체는 불변이므로, 많은 문자열을 함께 이어붙이면 각 이어붙이기가 새 객체를 생성하기 때문에 비효율적입니다. 일반적으로, 총 실행 시간 비용은 전체 문자열 길이의 제곱에 비례합니다.

많은 `str` 객체를 누적하기 위해, 권장되는 관용구는 객체를 리스트에 배치하고 마지막에 `str.join()` 을 호출하는 것입니다:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(또 다른 합리적으로 효율적인 관용구는 `io.StringIO`를 사용하는 것입니다)

많은 `bytes` 객체를 누적하기 위해, 권장되는 관용구는 제자리 이어붙이기(`+=` 연산자)을 사용하여 `bytearray` 객체를 확장하는 것입니다:

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

2.5 시퀀스 (튜플/리스트)

2.5.1 튜플과 리스트를 어떻게 변환합니까?

형 생성자 `tuple(seq)` 는 임의의 시퀀스(실제로는, 모든 이터러블)를 같은 순서로 같은 항목을 가진 튜플로 변환합니다.

예를 들어, `tuple([1, 2, 3])` 은 `(1, 2, 3)` 을 산출하고 `tuple('abc')` 는 `('a', 'b', 'c')` 를 산출합니다. 인자가 튜플이면 사본을 만들지 않고 같은 객체를 반환하므로, 객체가 이미 튜플인지 확실하지 않을 때 `tuple()` 을 호출하는 것이 저렴합니다.

형 생성자 `list(seq)` 는 임의의 시퀀스나 이터러블을 같은 순서로 같은 항목이 있는 리스트로 변환합니다. 예를 들어, `list((1, 2, 3))` 은 `[1, 2, 3]` 을 산출하고 `list('abc')` 는 `['a', 'b', 'c']` 를 산출합니다. 인자가 리스트이면, `seq[:]` 와 같이 사본을 만듭니다.

2.5.2 음수 인덱스는 무엇입니까?

파이썬 시퀀스는 양수와 음수로 인덱싱됩니다. 양수의 경우 0은 첫 번째 인덱스이고 1은 두 번째 인덱스이고 이런 식으로 계속됩니다. 음수 인덱스의 경우 -1은 마지막 인덱스이고 -2는 끝에서 두 번째 인덱스이고 이런 식으로 계속됩니다. `seq[-n]` 을 `seq[len(seq)-n]` 과 같다고 생각하십시오.

음수 인덱스를 사용하면 매우 편리할 수 있습니다. 예를 들어 `s[:-1]` 은 마지막 문자를 제외한 문자열의 모든 것인데, 문자열에서 후행 줄 바꿈을 제거하는 데 유용합니다.

2.5.3 시퀀스를 역순으로 이터레이트 하려면 어떻게 합니까?

`reversed()` 내장 함수를 사용하십시오:

```
for x in reversed(sequence):
    ... # do something with x ...
```

이것은 원본 시퀀스에는 영향을 미치지 않지만, 이터레이트 할 뒤집힌 순서의 새 사본을 만듭니다.

2.5.4 리스트에서 중복을 어떻게 제거합니까?

이 작업을 수행하는 여러 가지 방법에 대한 긴 논의는 파이썬 요리책을 참조하십시오:

<https://code.activestate.com/recipes/52560/>

리스트 순서를 바꿔도 상관없다면, 리스트를 정렬한 다음 리스트 끝에서 스캔하면서 중복 항목을 삭제하십시오:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

리스트의 모든 요소가 집합 키로 사용될 수 있다면 (즉, 모두 해시 가능이면) 이것이 종종 더 빠릅니다

```
mylist = list(set(mylist))
```

이것은 리스트를 집합으로 변환하여, 중복을 제거한 다음, 리스트로 되돌립니다.

2.5.5 리스트에서 여러 항목을 어떻게 제거합니까?

중복 제거와 마찬가지로, 삭제 조건을 사용하여 명시적으로 역순으로 이터레이션 하는 것도 한 가지 가능성이 있습니다. 그러나, 묵시적이나 명시적 순방향 이터레이션으로 슬라이스 치환을 사용하기가 더 쉽습니다. 다음은 세 가지 변형입니다.:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

리스트 컴프리헨션이 아마 가장 빠릅니다.

2.5.6 파이썬에서 어떻게 배열을 만드나요?

리스트를 사용하십시오:

```
["this", 1, "is", "an", "array"]
```

리스트는 시간 복잡성 면에서 C나 파스칼(Pascal) 배열과 동등합니다; 가장 큰 차이점은 파이썬 리스트에 다양한 형의 객체가 포함될 수 있다는 것입니다.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that `NumPy` and other third party packages define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate *cons cells* using tuples:

```
lisp_list = ("like", ("this", ("example", None) ) )
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of a Lisp *car* is `lisp_list[0]` and the analogue of *cdr* is `lisp_list[1]`. Only do this if you're sure you really need to, because it's usually a lot slower than using Python lists.

2.5.7 다차원 리스트를 어떻게 만듭니까?

다음과 같이 다차원 배열을 만들려고 했을 것입니다:

```
>>> A = [[None] * 2] * 3
```

인쇄하면 올바르게 보입니다:

```
>>> A
[[None, None], [None, None], [None, None]]
```

그러나 값을 대입하면, 여러 위치에 나타납니다:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

그 이유는 *로 리스트를 복제해도 복사본을 만들지 않고 기존 객체에 대한 참조만 만들기 때문입니다. *3은 길이 2의 같은 리스트에 대한 3개의 참조를 포함하는 리스트를 만듭니다. 한 행에 대한 변경 사항은 모든 행에 나타나는데, 거의 확실히 여러분이 원하는 것은 아닙니다.

제안된 방법은 원하는 길이의 리스트를 먼저 만든 다음 새로 만든 리스트로 각 요소를 채우는 것입니다:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

그러면 길이가 2인 3개의 다른 리스트를 포함하는 리스트가 생성됩니다. 리스트 컴프리헨션도 사용할 수 있습니다:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Or, you can use an extension that provides a matrix datatype; NumPy is the best known.

2.5.8 How do I apply a method or function to a sequence of objects?

To call a method or function and accumulate the return values in a list, a *list comprehension* is an elegant solution:

```
result = [obj.method() for obj in mylist]

result = [function(obj) for obj in mylist]
```

To just run the method or function without saving the return values, a plain `for` loop will suffice:

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

2.5.9 덧셈은 작동하는데, 왜 `a_tuple[i] += ['item']`이 예외를 일으킵니까?

이는 증분 대입 연산자가 대입 연산자라는 사실과 파이썬에서 가변 객체와 불변 객체의 차이점이 결합하기 때문입니다.

이 논의는 증분 대입 연산자가 가변 객체를 가리키는 튜플의 요소에 적용될 때 일반적으로 적용되지만, 우리는 `list`와 `+=`를 예제로 사용합니다.

다음과 같이 작성한다면:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

예외의 이유는 명확합니다: 1이 `a_tuple[0]`이 가리키는 객체(1)에 더해져서, 결과 객체 2를 생성하지만, 계산 결과 2를 튜플의 요소 0에 대입하려고 하면, 튜플의 요소가 가리키는 것을 변경할 수 없기 때문에 예외가 발생합니다.

수면 아래에서, 이 증분 대입문이 하는 일은 대략 다음과 같습니다:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

튜플은 불변이므로, 연산의 대입 부분이 예외를 발생시킵니다.

다음과 같이 작성하면:

```
>>> a_tuple = ('foo', 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

예외는 조금 더 놀랍습니다, 더 놀라운 것은 예외가 있었지만 더하기가 동작했다는 사실입니다:

```
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__()` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, `__iadd__()` is equivalent to calling `extend()` on the list and returning the list. That's why we say that for lists, `+=` is a “shorthand” for `list.extend()`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

이것은 다음과 동등합니다:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

`a_list`가 가리키는 객체가 변경되었고, 변경된 객체에 대한 포인터가 다시 `a_list`에 대입됩니다. 대입의 최종 결과는 no-op인데, `a_list`가 이전에 가리키고 있던 것과 같은 객체에 대한 포인터이기 때문입니다, 하지만 대입은 여전히 일어납니다.

따라서, 우리의 튜플 예제에서 일어나는 일은 다음과 동등합니다:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

The `__iadd__()` succeeds, and thus the list is extended, but even though `result` points to the same object that `a_tuple[0]` already points to, that final assignment still results in an error, because tuples are immutable.

2.5.10 복잡한 정렬을 하고 싶습니다: 파이썬에서 Schwartzian 변환을 할 수 있습니까?

Perl 커뮤니티의 Randal Schwartz에 의한 이 기법은 리스트의 각 요소를 각 요소를 “정렬 값”에 매핑하는 메트릭으로 정렬합니다. 파이썬에서는, `list.sort()` 메서드의 `key` 인자를 사용하십시오:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

2.5.11 한 리스트를 다른 리스트의 값으로 정렬하려면 어떻게 해야 합니까?

그것들을 튜플의 이터레이터로 병합하고, 결과 리스트를 정렬한 다음, 원하는 요소를 선택하십시오.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[('I'm', 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

2.6 객체

2.6.1 클래스는 무엇입니까?

클래스는 `class` 문을 실행하여 만든 특정 객체 형입니다. 클래스 객체는 인스턴스 객체를 만들기 위한 주형으로 사용되며, 데이터형과 관련된 데이터(어트리뷰트)와 코드(메서드)를 모두 내장합니다.

클래스는 베이스 클래스라고 하는 하나 이상의 다른 클래스를 기반으로 할 수 있습니다. 그러면 베이스 클래스의 어트리뷰트와 메서드를 상속합니다. 이는 상속을 통해 객체 모델을 점진적으로 재정의할 수 있도록 합니다. 우편함에 대한 기본 접근자 메서드를 제공하는 일반 `Mailbox` 클래스와 다양한 특정 사서함 형식을 처리하는 `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox`와 같은 서브 클래스가 있을 수 있습니다.

2.6.2 메서드는 무엇입니까?

메서드는 일반적으로 `x.name(arguments...)`로 호출하는 어떤 객체 `x`의 함수입니다. 메서드는 클래스 정의 내에서 함수로 정의됩니다:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

2.6.3 self는 무엇입니까?

`self`는 단지 메서드의 첫 번째 인자를 위한 관례적 이름입니다. `meth(self, a, b, c)`로 정의된 메서드는 정의가 등장한 클래스의 어떤 인스턴스 `x`에 대해 `x.meth(a, b, c)`로 호출되어야 합니다; 호출된 메서드는 `meth(x, a, b, c)`처럼 호출되었다고 생각합니다.

메서드 정의와 호출에서 '`self`'를 명시적으로 사용해야 하는 이유는 무엇입니까? 도 참조하십시오.

2.6.4 객체가 주어진 클래스나 그 서브 클래스의 인스턴스인지 어떻게 확인합니까?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, float, complex))`.

Note that `isinstance()` also checks for virtual inheritance from an *abstract base class*. So, the test will return `True` for a registered class even if hasn't directly or indirectly inherited from it. To test for "true inheritance", scan the *MRO* of the class:

```
from collections.abc import Mapping

class P:
    pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)           # direct
True
>>> isinstance(c, P)           # indirect
True
>>> isinstance(c, Mapping)     # virtual
True

# Actual inheritance chain
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

대부분의 프로그램은 사용자 정의 클래스에서 `isinstance()` 를 자주 사용하지 않음에 유의하십시오. 클래스를 직접 개발하고 있다면, 더 적절한 객체 지향 스타일은 객체의 클래스를 확인하고 클래스에 따라 다른 작업을 수행하는 대신 특정 동작을 캡슐화하는 클래스의 메서드를 정의하는 것입니다. 예를 들어, 무언가를 수행하는 함수가 있다면:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ... # code to search a mailbox
    elif isinstance(obj, Document):
        ... # code to search a document
    elif ...
```

더 나은 접근법은 모든 클래스에서 `search()` 메서드를 정의하고 단지 그것을 호출하는 것입니다:

```
class Mailbox:
    def search(self):
        ... # code to search a mailbox

class Document:
    def search(self):
        ... # code to search a document

obj.search()
```

2.6.5 위임이란 무엇입니까?

위임(delegation)은 객체 지향 기법(디자인 패턴이라고도 합니다)입니다. `x` 객체가 있고 메서드 중 하나의 동작을 변경하고 싶다고 가정해 봅시다. 변경하려는 메서드의 새로운 구현을 제공하고 다른 모든 메서드를 `x`의 해당 메서드에 위임하는 새 클래스를 만들 수 있습니다.

파이썬 프로그래머는 쉽게 위임을 구현할 수 있습니다. 예를 들어, 다음 클래스는 파일처럼 동작하지만, 기록되는 모든 데이터를 대문자로 변환하는 클래스를 구현합니다:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self._outfile.write()` method. All other methods are delegated to the underlying `self._outfile` object. The delegation is accomplished via the `__getattr__()` method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a `__setattr__()` method too, and it must do so carefully. The basic implementation of

`__setattr__()` is roughly equivalent to the following:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Many `__setattr__()` implementations call `object.__setattr__()` to set an attribute on `self` without causing infinite recursion:

```
class X:
    def __setattr__(self, name, value):
        # Custom logic here...
        object.__setattr__(self, name, value)
```

Alternatively, it is possible to set attributes by inserting entries into `self.__dict__` directly.

2.6.6 How do I call a method defined in a base class from a derived class that extends it?

내장 `super()` 함수를 사용하십시오:

```
class Derived(Base):
    def meth(self):
        super().meth()  # calls Base.meth
```

In the example, `super()` will automatically determine the instance from which it was called (the `self` value), look up the *method resolution order* (MRO) with `type(self).__mro__`, and return the next in line after `Derived` in the MRO: `Base`.

2.6.7 베이스 클래스를 쉽게 변경할 수 있도록 코드를 구성하려면 어떻게 해야 합니까?

베이스 클래스를 별칭에 대입하고 별칭에서 파생할 수 있습니다. 그러면 별칭에 대입된 값만 변경하면 됩니다. 또한 이 트릭은 사용할 베이스 클래스를 동적으로 (예를 들어 자원의 가용성에 따라) 결정하려는 경우에도 유용합니다. 예:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

2.6.8 정적 클래스 데이터와 정적 클래스 메서드를 만들려면 어떻게 해야 합니까?

(C++나 Java의 의미에서) 정적 데이터와 정적 메서드 모두 파이썬에서 지원됩니다.

정적 데이터의 경우, 단순히 클래스 어트리뷰트를 정의하십시오. 어트리뷰트에 새 값을 대입하려면, 대입에서 클래스 이름을 명시적으로 사용해야 합니다:

```
class C:
    count = 0  # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def getcount(self):
    return C.count # or return self.count
```

c 자체나 c.__class__에서 C로 돌아가는 베이스 클래스 검색 경로에 놓인 일부 클래스에 의해 재정의되지 않는 한, c.count는 isinstance(c, C)가 성립하는 모든 c에 대해 C.count를 참조합니다.

주의: C의 메서드 내에서, self.count = 42와 같은 대입은 self의 자체 디렉터리에 “count”라는 새롭고 관련이 없는 인스턴스를 만듭니다. 클래스 정적 데이터 이름의 재연결은 항상 메서드 내부에 있는지에 관계없이 클래스를 지정해야 합니다:

```
C.count = 314
```

정적 메서드도 가능합니다:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

그러나, 정적 메서드의 효과를 얻는 훨씬 간단한 방법은 단순한 모듈 수준 함수를 사용하는 것입니다:

```
def getcount():
    return C.count
```

여러분의 코드가 모듈 당 하나의 클래스 (또는 밀접하게 관련된 클래스 계층 구조)를 정의하도록 구조화되었다면, 이것이 원하는 캡슐화를 제공합니다.

2.6.9 파이썬에서 생성자(또는 메서드)를 어떻게 재정의할 수 있습니까?

이 답변은 실제로 모든 메서드에 적용되지만, 질문은 일반적으로 생성자 문맥에서 가장 먼저 나옵니다.

C++에서는 다음과 같이 작성합니다

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

파이썬에서는 기본 인자를 사용하여 모든 경우를 다루는 단일 생성자를 작성해야 합니다. 예를 들면:

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

이것은 완전히 동등하지는 않지만, 실제로는 아주 가깝습니다.

가변 길이 인자 목록을 시도할 수도 있습니다, 예를 들어:

```
def __init__(self, *args):
    ...
```

같은 접근법이 모든 메서드 정의에서도 동작합니다.

2.6.10 `__spam`을 사용하려고 하는데 `_SomeClassName__spam`에 대한 에러가 발생합니다.

이중 선행 밑줄이 있는 변수 이름은 클래스 비공개(private) 변수를 정의하는 간단하지만, 효과적인 방법을 제공하기 위해 “뒤섞입니다(mangled)”. `__spam` 형식(적어도 두 개의 선행 밑줄, 최대 하나의 후행 밑줄)의 모든 식별자는 `_classname__spam`으로 텍스트 대체되는데, 여기서 `classname`은 모든 선행 밑줄이 제거된 현재 클래스 이름입니다.

The identifier can be used unchanged within the class, but to access it outside the class, the mangled name must be used:

```
class A:
    def __one(self):
        return 1
    def two(self):
        return 2 * self.__one()

class B(A):
    def three(self):
        return 3 * self._A__one()

four = 4 * A()._A__one()
```

In particular, this does not guarantee privacy since an outside user can still deliberately access the private attribute; many Python programmers never bother to use private variable names at all.

➡ 더 보기

The private name mangling specifications for details and special cases.

2.6.11 내 클래스는 `__del__`을 정의하지만 객체를 삭제할 때 호출되지 않습니다.

몇 가지 가능한 이유가 있습니다.

The `del` statement does not necessarily call `__del__()` – it simply decrements the object’s reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you’re trying to reproduce a problem. Worse, the order in which object’s `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

Despite the cycle collector, it’s still a good idea to define an explicit `close()` method on objects to be called whenever you’re done with them. The `close()` method can then remove attributes that refer to subobjects. Don’t call `__del__()` directly – `__del__()` should call `close()` and `close()` should make sure that it can be called more than once for the same object.

순환 참조를 피하는 또 다른 방법은 `weakref` 모듈을 사용하는 것입니다. 이 모듈은 참조 횟수를 늘리지 않고 객체를 가리킬 수 있도록 합니다. 예를 들어, 트리 자료 구조는 부모와 형제 참조에 대해 약한 참조를 사용해야 합니다 (이런 것들이 필요하다면!).

Finally, if your `__del__()` method raises an exception, a warning message is printed to `sys.stderr`.

2.6.12 주어진 클래스의 모든 인스턴스 목록을 어떻게 얻습니까?

파이썬은 클래스(또는 내장형)의 모든 인스턴스를 추적하지 않습니다. 클래스 생성자가 각 인스턴스에 대한 약한 참조 리스트를 유지하여 모든 인스턴스를 추적하도록 프로그래밍 할 수 있습니다.

2.6.13 `id()`의 결과가 고유하지 않은 것처럼 보이는 이유는 무엇입니까?

`id()` 내장은 객체 수명 동안 고유하도록 보장되는 정수를 반환합니다. CPython에서는 이것이 객체의 메모리 주소이므로, 객체가 메모리에서 삭제된 후 새로 만들어진 다음 객체가 메모리의 같은 위치에 할당되는 경우가 자주 발생합니다. 다음과 같이 예시할 수 있습니다:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

두 개의 `id`는 다른 정수 객체에 속하는데, `id()` 호출 실행 앞에 만들어지고, 호출 직후 삭제됩니다. `id`를 검사하려는 객체가 여전히 살아 있도록 하려면, 그 객체에 대한 다른 참조를 만드십시오:

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

2.6.14 `is` 연산자를 사용한 아이덴티티 검사는 언제 신뢰할 수 있습니까?

`is` 연산자는 객체 아이덴티티를 검사합니다. 검사 `a is b`는 `id(a) == id(b)`와 동등합니다.

아이덴티티 검사의 가장 중요한 속성은 객체가 항상 자신과 동일하고 `a is a`는 항상 `True`를 반환한다는 것입니다. 아이덴티티 검사는 일반적으로 동등성 검사보다 빠릅니다. 동등성 검사와 달리, 아이덴티티 테스트는 불리언 `True`나 `False`를 반환함이 보장됩니다.

그러나, 객체 아이덴티티가 보장될 때 아이덴티티 검사가 동등성 검사를 대체할 수 있습니다. 일반적으로, 아이덴티티가 보장되는 세 가지 상황이 있습니다:

- 1) Assignments create new names but do not change object identity. After the assignment `new = old`, it is guaranteed that `new is old`.
- 2) Putting an object in a container that stores object references does not change object identity. After the list assignment `s[0] = x`, it is guaranteed that `s[0] is x`.
- 3) If an object is a singleton, it means that only one instance of that object can exist. After the assignments `a = None` and `b = None`, it is guaranteed that `a is b` because `None` is a singleton.

대부분의 다른 상황에서는 아이덴티티 검사가 권장되지 않으며 동등성 테스트가 선호됩니다. 특히, 싱글톤이 보장되지 않는 `int`와 `str`과 같은 상수를 확인하는 데 아이덴티티 검사를 사용해서는 안 됩니다:

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

마찬가지로, 가변 컨테이너의 새 인스턴스는 절대 동일하지 않습니다:

```
>>> a = []
>>> b = []
>>> a is b
False
```

표준 라이브러리 코드에서, 아이덴티티 검사를 올바르게 사용하는 몇 가지 일반적인 패턴을 볼 수 있습니다:

- 1) As recommended by [PEP 8](#), an identity test is the preferred way to check for `None`. This reads like plain English in code and avoids confusion with other objects that may have boolean values that evaluate to false.
- 2) Detecting optional arguments can be tricky when `None` is a valid input value. In those situations, you can create a singleton sentinel object guaranteed to be distinct from other objects. For example, here is how to implement a method that behaves like `dict.pop()`:

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
    if default is _sentinel:
        raise KeyError(key)
    return default
```

- 3) Container implementations sometimes need to augment equality tests with identity tests. This prevents the code from being confused by objects such as `float('NaN')` that are not equal to themselves.

For example, here is the implementation of `collections.abc.Sequence.__contains__()`:

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

2.6.15 How can a subclass control what data is stored in an immutable instance?

When subclassing an immutable type, override the `__new__()` method instead of the `__init__()` method. The latter only runs *after* an instance is created, which is too late to alter data in an immutable instance.

All of these immutable classes have a different signature than their parent class:

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

The classes can be used like this:

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

2.6.16 How do I cache method calls?

The two principal tools for caching methods are `functools.cached_property()` and `functools.lru_cache()`. The former stores results at the instance level and the latter at the class level.

The *cached_property* approach only works with methods that do not take any arguments. It does not create a reference to the instance. The cached method result will be kept only as long as the instance is alive.

The advantage is that when an instance is no longer used, the cached method result will be released right away. The disadvantage is that if instances accumulate, so too will the accumulated method results. They can grow without bound.

The *lru_cache* approach works with methods that have *hashable* arguments. It creates a reference to the instance unless special efforts are made to pass in weak references.

The advantage of the least recently used algorithm is that the cache is bounded by the specified *maxsize*. The disadvantage is that instances are kept alive until they age out of the cache or until the cache is cleared.

This example shows the various techniques:

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
        "Rainfall on a given date"
        # Depends on the station_id, date, and units.
```

The above example assumes that the *station_id* never changes. If the relevant instance attributes are mutable, the *cached_property* approach can't be made to work because it cannot detect changes to the attributes.

To make the *lru_cache* approach work when the *station_id* is mutable, the class needs to define the `__eq__()` and `__hash__()` methods so that the cache can detect relevant attribute updates:

```
class Weather:
    "Example with a mutable station identifier"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

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

def change_station(self, station_id):
    self.station_id = station_id

def __eq__(self, other):
    return self.station_id == other.station_id

def __hash__(self):
    return hash(self.station_id)

@lru_cache(maxsize=20)
def historic_rainfall(self, date, units='cm'):
    'Rainfall on a given date'
    # Depends on the station_id, date, and units.

```

2.7 모듈

2.7.1 .pyc 파일을 어떻게 만듭니까?

모듈이 처음 임포트 될 때 (또는 현재 컴파일된 파일이 만들어진 후 소스 파일이 변경되었을 때) 컴파일된 코드를 포함하는 .pyc 파일은 .py 파일을 포함하는 디렉터리의 __pycache__ 서브 디렉터리에 만들어져야 합니다. .pyc 파일은 .py 파일과 같은 이름으로 시작하고, .pyc로 끝나며, 파일을 만든 특정 python 바이너리 파일에 종속되는 중간 구성 요소를 갖는 파일명을 갖습니다. (자세한 내용은 [PEP 3147](#)을 참조하십시오.)

.pyc 파일이 만들어지지 않을 수 있는 한 가지 이유는 소스 파일이 포함된 디렉터리의 권한 문제입니다. 즉, __pycache__ 서브 디렉터리를 만들 수 없다는 뜻입니다. 예를 들어, 한 사용자로 개발했지만 다른 사용자로 실행하는 경우에 이런 일이 일어날 수 있습니다, 가령 웹 서버로 테스트하고 있을 때입니다.

PYTHONDONTWRITEBYTECODE 환경 변수가 설정되어 있지 않은 한, 모듈을 임포트 하고 파이썬이 __pycache__ 서브 디렉터리를 만들고 컴파일된 모듈을 그 서브 디렉터리에 쓸 수 있는 능력(권한, 여유 공간 등)이 있으면 .pyc 파일의 생성은 자동입니다.

최상위 스크립트에서 파이썬을 실행하는 것은 임포트로 간주하지 않으며 .pyc가 만들어지지 않습니다. 예를 들어, 다른 모듈 xyz.py를 임포트 하는 최상위 모듈 foo.py가 있을 때, (python foo.py를 셸 명령으로 입력하여) foo를 실행하면, xyz를 임포트 하기 때문에 xyz에 대해 .pyc가 만들어지지만, foo.py를 임포트 하지 않기 때문에 foo에 대해서는 .pyc 파일이 만들어지지 않습니다.

foo에 대한 .pyc 파일을 만들 필요가 있으면 - 즉, 임포트 되지 않는 모듈에 대한 .pyc 파일을 만들려면 - py_compile과 compileall 모듈을 사용할 수 있습니다.

py_compile 모듈은 임의의 모듈을 수동으로 컴파일할 수 있습니다. 한 가지 방법은 해당 모듈에서 compile() 함수를 대화식으로 사용하는 것입니다:

```

>>> import py_compile
>>> py_compile.compile('foo.py')

```

이것은 .pyc를 foo.py와 같은 위치에 있는 __pycache__ 서브 디렉터리에 기록합니다 (또는 선택적 매개 변수 cfile로 이를 재정의할 수 있습니다).

compileall 모듈을 사용하여 디렉터리의 모든 파일을 자동으로 컴파일할 수도 있습니다. compileall.py를 실행하고 컴파일할 파이썬 파일이 포함된 디렉터리의 경로를 제공하여 셸 프롬프트에서 이를 수행할 수 있습니다:

```
python -m compileall .
```

2.7.2 현재 모듈 이름을 어떻게 찾습니까?

모듈은 사전 정의된 전역 변수 `__name__`을 봄으로써 모듈 자신의 이름을 찾을 수 있습니다. 값이 `'__main__'`이면, 프로그램이 스크립트로 실행 중입니다. 일반적으로 모듈을 임포트 해서 사용하는 많은 모듈은 명령 줄 인터페이스나 자체 테스트를 제공하며, `__name__`을 확인한 후에 만 이 코드를 실행합니다:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

2.7.3 서로 임포트 하는 모듈을 어떻게 만들 수 있습니까?

다음 모듈이 있다고 가정하십시오:

foo.py:

```
from bar import bar_var
foo_var = 1
```

bar.py:

```
from foo import foo_var
bar_var = 2
```

문제는 인터프리터가 다음 단계를 수행한다는 것입니다:

- main imports foo
- Empty globals for foo are created
- foo is compiled and starts executing
- foo imports bar
- Empty globals for bar are created
- bar is compiled and starts executing
- bar imports foo (which is a no-op since there already is a module named foo)
- The import mechanism tries to read `foo_var` from `foo` globals, to set `bar.foo_var = foo.foo_var`

마지막 단계가 실패합니다. 파이썬이 아직 `foo`를 인터프리트 하는 것을 완료하지 않았고 `foo`의 전역 기호 딕셔너리가 여전히 비어 있기 때문입니다.

`import foo`를 사용하고 전역 코드에서 `foo.foo_var`에 액세스하려고 할 때도 같은 일이 일어납니다.

이 문제에 대해 가능한 (최소한) 세 가지 해결 방법이 있습니다.

Guido van Rossum은 `from <module> import ...`을 아예 사용하지 말고, 모든 코드를 함수 내에 배치할 것을 권장합니다. 전역 변수와 클래스 변수의 초기화는 상수나 내장 함수만 사용해야 합니다. 이것은 임포트 된 모듈의 모든 것이 `<module>.<name>`으로 참조됨을 의미합니다.

Jim Roskind는 각 모듈에서 다음 순서로 단계를 수행할 것을 제안합니다:

- 내보내기 (전역, 함수 및 베이스 클래스를 임포트 할 필요가 없는 클래스)
- import 문
- 활성 코드 (임포트 된 값에서 초기화된 전역 포함).

Van Rossum doesn't like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs는 처음부터 재귀 임포트가 필요하지 않도록 코드를 재구성할 것을 권합니다.

이 해결 방법들은 상호 배타적이지 않습니다.

2.7.4 `__import__`('x.y.z')는 <module 'x' >를 반환합니다; z를 어떻게 얻습니까?

`importlib`의 편의 함수 `import_module()` 을 대신 사용하는 곳을 고려하십시오:

```
z = importlib.import_module('x.y.z')
```

2.7.5 임포트 된 모듈을 편집하고 다시 임포트 할 때, 변경 사항이 표시되지 않습니다. 왜 이런 일이 발생합니까?

효율성뿐만 아니라 일관성의 이유로, 파이썬은 모듈을 처음 임포트 할 때만 모듈 파일을 읽습니다. 그렇지 않으면, 각 모듈이 같은 기본 모듈을 임포트 하는 많은 모듈로 구성된 프로그램에서, 기본 모듈을 여러 번 구문 분석하고 다시 구문 분석하게 됩니다. 변경된 모듈을 강제로 다시 읽으려면, 다음과 같이 하십시오:

```
import importlib
import modname
importlib.reload(modname)
```

경고: 이 기법은 100% 확실하지 않습니다. 특히, 다음과 같은 문장을 포함하는 모듈은

```
from modname import some_objects
```

임포트 된 객체의 이전 버전으로 계속 작업합니다. 모듈에 클래스 정의가 포함되면, 새 클래스 정의를 사용하도록 기존 클래스 인스턴스가 갱신되지 않습니다. 이것은 다음과 같은 역설적인 동작으로 이어집니다:

```
>>> import importlib
>>> import cls
>>> c = cls.C()                                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)                       # isinstance is false!?!
False
```

클래스 객체의 “아이덴티티”를 인쇄하면 문제의 본질이 분명해집니다:

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```


3.1 파이썬은 왜 문장의 그룹화에 들여쓰기를 사용합니까?

Guido van Rossum은 그룹화에 들여쓰기를 사용하는 것이 매우 우아하고 일반적인 파이썬 프로그램의 명확성에 크게 기여한다고 믿습니다. 대부분의 사람은 시간이 좀 흐른 후에 이 기능을 사랑하는 법을 배웁니다.

시작/끝 괄호가 없기 때문에 구문 분석기와 사람 독자가 인식하는 그룹 간에 불일치가 있을 수 없습니다. 때때로 C 프로그래머는 다음과 같은 코드 조각을 만나게 됩니다:

```
if (x <= y)
    x++;
    y--;
z++;
```

조건이 참이면 `x++` 문만 실행되지만, 들여쓰기는 많은 사람이 그렇지 않다고 믿게 만듭니다. 경험 많은 C 프로그래머조차도 `x > y` 일 때도 `y`가 감소하는 이유를 궁금해하면서 오래 들여다볼 때가 있습니다.

시작/끝 괄호가 없기 때문에, 파이썬은 코딩 스타일 충돌이 훨씬 적습니다. C에서는 중괄호를 배치하는 여러 가지 방법이 있습니다. 특정 스타일을 사용하여 코드를 읽고 쓰는 데 익숙해지면, 다른 스타일로 읽을 (또는 작성해야 할) 때 다소 불편함을 느끼는 것은 정상입니다.

많은 코딩 스타일은 시작/끝 괄호를 그 자신만의 줄에 배치합니다. 이로 인해 프로그램이 상당히 길어지고 귀중한 화면 공간이 낭비되어, 프로그램을 조망하기가 더 어려워집니다. 이상적으로는, 함수가 한 화면에 맞아야 합니다 (가령, 20-30줄). 20줄의 파이썬은 C의 20줄보다 훨씬 더 많은 작업을 수행할 수 있습니다. 이것은 시작/끝 괄호가 필요 없기 때문 만은 아닙니다만 - 선언이 없는 것과 고수준의 데이터형도 기여합니다 - 들여쓰기 기반 문법은 확실히 도움이 됩니다.

3.2 간단한 산술 연산으로 이상한 결과가 나오는 이유는 무엇입니까?

다음 질문을 보십시오.

3.3 부동 소수점 계산이 왜 그렇게 부정확합니까?

사용자는 종종 다음과 같은 결과에 놀라게 됩니다:

```
>>> 1.2 - 1.0
0.19999999999999996
```

그리고 이것을 파이썬의 버그라고 생각합니다. 그렇지 않습니다. 이것은 파이썬과 거의 관련이 없으며, 하부 플랫폼이 부동 소수점 숫자를 처리하는 방법과 훨씬 더 관련이 있습니다.

CPython의 `float` 형은 저장을 위해 C `double`을 사용합니다. `float` 객체의 값은 고정 정밀도(일반적으로 53비트)의 이진 부동 소수점에 저장되고 파이썬은 부동 소수점 연산을 수행하는 데 C 연산을 사용하며, 이는 다시 프로세서의 하드웨어 구현에 의존합니다. 즉, 부동 소수점 연산에 관한 한, 파이썬은 C와 Java를 포함한 많은 널리 알려진 언어들처럼 작동합니다.

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating point. For example, after:

```
>>> x = 1.2
```

이후에, `x`에 대해 저장된 값은 10진수 값 1.2에 대한(매우 좋은) 근사치이지만, 정확히 같지는 않습니다. 일반적인 기계에서 실제 저장되는 값은 다음과 같습니다:

```
1.0011001100110011001100110011001100110011001100110011001100110011 (binary)
```

이것은 정확하게는 다음과 같습니다:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

53비트의 일반적인 정밀도는 파이썬 부동 소수점에 15-16자리의 10진수 정확도를 제공합니다.

For a fuller explanation, please see the floating-point arithmetic chapter in the Python tutorial.

3.4 파이썬 문자열이 불변인 이유는 무엇입니까?

몇 가지 장점이 있습니다.

하나는 성능입니다: 문자열이 불변임을 안다는 것은 만들 때 이를 위한 공간을 할당할 수 있다는 것을 의미하며, 스토리지 요구 사항은 고정되고 변경되지 않습니다. 이것은 또한 튜플과 리스트를 구분하는 이유 중 하나입니다.

또 다른 장점은 파이썬의 문자열을 숫자만큼 “기본적”으로 간주한다는 것입니다. 어떤 방법도 값 8을 다른 것으로 변경하지 않으며, 파이썬에서는 어떤 방법도 문자열 “eight”을 다른 것으로 변경하지 않습니다.

3.5 메서드 정의와 호출에서 ‘self’를 명시적으로 사용해야 하는 이유는 무엇입니까?

아이디어는 Modula-3에서 빌렸습니다. 여러 가지 이유로 매우 유용합니다.

첫째, 로컬 변수 대신 메서드나 인스턴스 어트리뷰트를 사용하고 있다는 것이 더 분명합니다. `self.x`나 `self.meth()`를 읽으면 클래스 정의를 기억하지 못하더라도 인스턴스 변수나 메서드가 사용된다는 것을 분명히 알 수 있습니다. C++에서는, 지역 변수 선언이 없는 것으로 구분할 수 있습니다(전역은 드물거나 쉽게 인식할 수 있다고 가정할 때) - 하지만 파이썬에서는, 지역 변수 선언이 없어서, 확실히 하려면 클래스 정의를 찾아야 합니다. 일부 C++와 Java 코딩 표준에서는 인스턴스 어트리뷰트에 `m_` 접두어를 요청합니다, 따라서 이러한 명시성은 이런 언어들에서도 여전히 유용합니다.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator - in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

마지막으로, 인스턴스 변수의 경우 대입과 관련된 문법 문제를 해결합니다: 파이썬의 지역 변수는(정의상!) 함수 본문에서 값이 대입되는(그리고 전역으로 명시적으로 선언되지 않은) 변수이기 때문에, 인터프리터에게 대입이 지역 변수가 아니라 인스턴스 변수에 대한 대입이라는 것을 알릴 방법이 있어야 하고, (효율성의 측면에서) 구문적이면 좋습니다. C++는 선언을 통해 이 작업을 수행하지만, 파이썬에는 선언이 없어서 이러한 목적으로만 문법을 도입해야 하는 것은 유감입니다. 명시적 `self.var`를 사용하면 이 문제가 잘 해결됩니다. 마찬가지로, 인스턴스 변수를 사용하는 경우, `self.var`라고 써야 한다는 것은, 메서드 내에서

정규화되지 않은 이름에 대한 참조가 인스턴스의 디렉터리를 검색할 필요가 없음을 의미합니다. 다시 말해, 지역 변수와 인스턴스 변수는 두 개의 서로 다른 이름 공간에 있으며, 사용할 이름 공간을 파이썬에 알려야 합니다.

3.6 표현식에서 대입을 사용할 수 없는 이유는 무엇입니까?

파이썬 3.8부터, 가능합니다!

Assignment expressions using the walrus operator `:=` assign a variable in an expression:

```
while chunk := fp.read(200):
    print(chunk)
```

자세한 정보는 [PEP 572](#)를 참조하십시오.

3.7 파이썬은 왜 일부 기능(예를 들어 `list.index()`)에는 메서드를 사용하고 다른 기능(예를 들어 `len(list)`)에는 함수를 사용합니까?

Guido가 말했듯이:

(a) 일부 연산의 경우, 전위 표기법(prefix notation)이 후위(postfix) 표기법보다 더 잘 읽힙니다 - 전위 (그리고 중위 infix!) 연산은 수학에서 오랜 전통을 가지고 있는데, 수학자가 문제에 대해 생각하는 데 시각적으로 도움이 되는 표기법을 좋아하는 곳입니다. $x*(a+b)$ 와 같은 공식을 $x*a + x*b$ 로 다시 작성하기 쉬운 것과 원시 OO 표기법을 사용하여 같은 작업을 수행할 때의 어색함을 비교해 보십시오.

(b) `len(x)` 라는 코드를 읽을 때 저는 무언가의 길이를 요구하고 있다는 것을 압니다. 이것은 저에게 두 가지를 알려줍니다: 결과는 정수이고, 인자는 일종의 컨테이너입니다. 반대로, `x.len()` 을 읽을 때, `x`가 인터페이스를 구현하거나 표준 `len()` 이 있는 클래스에서 상속하는 일종의 컨테이너라는 것을 이미 알고 있어야 합니다. 매핑을 구현하지 않는 클래스에 `get()` 이나 `keys()` 메서드가 있거나, 파일이 아닌 것에 `write()` 메서드가 있을 때 때때로 겪는 혼란을 보십시오.

—<https://mail.python.org/pipermail/python-3000/2006-November/004643.html>

3.8 `join()`이 리스트나 튜플 메서드가 아니라 문자열 메서드인 이유는 무엇입니까?

항상 `string` 모듈의 함수를 사용하여 제공되었던 것과 같은 기능을 제공하는 메서드가 추가된, 파이썬 1.6 부터 문자열은 다른 표준형과 훨씬 더 비슷해졌습니다. 이러한 새로운 메서드의 대부분은 널리 받아들여졌지만, 일부 프로그래머가 불편해하는 것은 다음과 같습니다:

```
", ".join(['1', '2', '4', '8', '16'])
```

이것은 다음과 같은 결과를 제공합니다:

```
"1, 2, 4, 8, 16"
```

이 사용법에 대해 두 가지 자주 나오는 반론이 있습니다.

첫 번째는 “문자열 리터럴(문자열 상수)의 메서드를 사용하는 것이 정말 보기 흉합니다”라는 줄이 따라옵니다. 이에 대한 대답은 그럴 수 있지만, 문자열 리터럴은 그저 고정된 값일 뿐이라는 것입니다. 메서드가 문자열에 연결된 이름에 허용된다면, 리터럴에서 사용할 수 없게 만드는 논리적 이유가 없습니다.

두 번째 이의는 보통 다음과 같이 표현됩니다: “나는 시퀀스가 멤버를 문자열 상수로 연결하라고 말하고 있습니다”. 슬프게도, 당신은 그렇지 않습니다. 어떤 이유로 `split()` 를 문자열 메서드로 사용하는 데 훨씬 덜 어려움을 겪는 것 같습니다, 그럴 때 다음과 같은 표현이

```
"1, 2, 4, 8, 16".split(", ")
```

문자열 리터럴에게 주어진 구분자(또는, 기본적으로, 임의의 공백 연속)로 구분된 하위 문자열을 반환하도록 하는 명령임을 쉽게 알 수 있기 때문입니다.

`join()` 은 문자열 메서드입니다. 사용 시 구분자 문자열에게 문자열 시퀀스를 이터레이트 하고 인접한 요소 사이에 자신을 삽입하도록 지시하기 때문입니다. 이 메서드는 사용자가 직접 정의할 수 있는 새 클래스를 포함하여, 시퀀스 객체에 대한 규칙을 따르는 모든 인자와 함께 사용할 수 있습니다. 바이트열과 `bytearray` 객체에 대해 유사한 메서드가 존재합니다.

3.9 예외는 얼마나 빠릅니까?

A `try/except` block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

이것은 딕셔너리에 거의 항상 키가 있을 것으로 예상했을 때만 의미가 있습니다. 그렇지 않으면, 다음과 같이 코딩했습니다:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

이 특정 경우에, `value = dict.setdefault(key, getvalue(key))` 를 사용할 수도 있지만, `getvalue()` 호출이 모든 경우에 평가되기 때문에 충분히 저렴한 경우에만 사용할 수 있습니다.

3.10 파이썬에 switch 나 case 문이 없는 이유는 무엇입니까?

In general, structured switch statements execute one block of code when an expression has a particular value or set of values. Since Python 3.10 one can easily match literal values, or constants within a namespace, with a `match ... case` statement. An older alternative is a sequence of `if... elif... elif... else`.

매우 많은 가능성 중에서 선택해야 하는 경우에는, case 값을 호출할 함수에 매핑하는 딕셔너리를 만들 수 있습니다. 예를 들면:

```
functions = {'a': function_1,
            'b': function_2,
            'c': self.method_1}

func = functions[value]
func()
```

객체에 대한 메서드 호출의 경우, `getattr()` 내장 함수를 사용하여 특정 이름을 가진 메서드를 꺼내어 더욱 단순화 할 수 있습니다:

```
class MyVisitor:
    def visit_a(self):
        ...

    def dispatch(self, value):
        method_name = 'visit_' + str(value)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
method = getattr(self, method_name)
method()
```

이 예제의 `visit_`와 같이 메서드 이름에 접두사를 사용하는 것이 좋습니다. 이러한 접두사가 없으면, 값이 신뢰할 수 없는 소스에서 오는 경우, 공격자가 객체의 모든 메서드를 호출할 수 있습니다.

Imitating switch with fallthrough, as with C's switch-case-default, is possible, much harder, and less needed.

3.11 OS별 스레드 구현에 의존하는 대신 인터프리터에서 스레드를 에뮬레이션할 수 없습니까?

답변 1: 불행히도, 인터프리터는 각 파이썬 스택 프레임에 대해 적어도 하나의 C 스택 프레임을 푸시합니다. 또한, 확장은 거의 임의의 순간에 파이썬으로 콜백 할 수 있습니다. 따라서, 전체 스레드 구현에는 C에 대한 스레드 지원이 필요합니다.

답변 2: 다행히, C 스택을 피하도록 완전히 재설계된 인터프리터 루프가 있는 [Stackless Python](#)이 있습니다.

3.12 람다 표현식이 문장을 포함할 수 없는 이유는 무엇입니까?

파이썬의 구문 프레임워크는 표현식 내부에 중첩된 문장을 처리할 수 없기 때문에 파이썬 람다 표현식은 문장을 포함할 수 없습니다. 그러나, 파이썬에서, 이것은 심각한 문제가 아닙니다. 기능을 추가하는 다른 언어의 람다 형식과 달리, 파이썬 람다는 함수를 정의하기에 너무 게으른 경우를 위한 줄임 표기법일 뿐입니다.

Functions are already first class objects in Python, and can be declared in a local scope. Therefore the only advantage of using a lambda instead of a locally defined function is that you don't need to invent a name for the function – but that's just a local variable to which the function object (which is exactly the same type of object that a lambda expression yields) is assigned!

3.13 파이썬을 기계 코드, C 또는 다른 언어로 컴파일 할 수 있습니까?

[Cython](#) compiles a modified version of Python with optional annotations into C extensions. [Nuitka](#) is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language.

3.14 파이썬은 메모리를 어떻게 관리합니까?

파이썬 메모리 관리의 세부 사항은 구현에 따라 다릅니다. 파이썬의 표준 구현인 [CPython](#)은 참조 카운팅을 사용하여 액세스할 수 없는 객체를 감지하고, 또 다른 메커니즘을 사용하여 참조 순환을 수집하고, 액세스할 수 없는 순환을 찾고 관련된 객체를 삭제하는 순환 감지 알고리즘을 주기적으로 실행합니다. gc 모듈은 가비지 수집을 수행하고, 디버깅 통계를 얻고, 수거기의 매개 변수를 조정하는 함수를 제공합니다.

Other implementations (such as [Jython](#) or [PyPy](#)), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

일부 파이썬 구현에서, 다음 코드(CPython에서는 괜찮습니다)는 아마도 파일 기술자의 소진을 일으킵니다:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Indeed, using CPython's reference counting and destructor scheme, each new assignment to `f` closes the previous file. With a traditional GC, however, those file objects will only get collected (and closed) at varying and possibly long intervals.

모든 파이썬 구현에서 작동하는 코드를 작성하려면, 명시적으로 파일을 닫거나 `with` 문을 사용해야 합니다; 다음은 메모리 관리 체계와 관계없이 작동합니다:


```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

3.15 CPython이 더 전통적인 가비지 수거 체계를 사용하지 않는 이유는 무엇입니까?

우선, 이것은 C 표준 기능이 아니라서 이식성이 없습니다. (예, 우리는 Boehm GC 라이브러리에 대해 알고 있습니다. 대부분의 일반 플랫폼용 (그들 전부는 아닙니다) 어셈블러 코드가 있고, 대체로 투명하지만, 완전히 투명하지는 않습니다; 파이썬이 이것으로 작동하도록 하려면 패치가 필요합니다.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard `malloc()` and `free()` with versions provided by the GC library, an application embedding Python may want to have its own substitute for `malloc()` and `free()`, and may not want Python's. Right now, CPython works with anything that implements `malloc()` and `free()` properly.

3.16 CPython이 종료될 때 모든 메모리가 해제되지 않는 이유는 무엇입니까?

파이썬 모듈의 전역 이름 공간에서 참조된 객체는 파이썬이 종료될 때 항상 할당 해제되지 않습니다. 순환 참조가 있으면 발생할 수 있습니다. C 라이브러리에 의해 할당된 해제가 불가능한 특정 메모리도 있습니다 (예를 들어 Purify와 같은 도구는 이에 대해 불평합니다). 그러나, 파이썬은 종료 시 메모리 정리에 적극적이며 모든 단일 객체를 파괴하려고 시도합니다.

할당 해제 시 파이썬이 특정 항목을 삭제하도록 강제하려면 `atexit` 모듈을 사용하여 해당 삭제를 강제하는 함수를 실행하십시오.

3.17 별도의 튜플과 리스트 데이터형이 있는 이유는 무엇입니까?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal *records* or C *structs*; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

튜플은 불변입니다. 즉, 일단 튜플이 만들어지면, 어느 요소도 새 값으로 바꿀 수 없습니다. 리스트는 가변이라서, 언제든지 리스트의 요소를 변경할 수 있습니다. 불변인 요소만 딕셔너리 키로 사용할 수 있어서, 리스트가 아니라 튜플만 키로 사용할 수 있습니다.

3.18 CPython에서 리스트는 어떻게 구현됩니까?

CPython의 리스트는 실제로는 가변 길이 배열입니다, Lisp 스타일의 연결 리스트(linked lists)가 아닙니다. 구현은 다른 객체에 대한 참조의 연속적인 배열을 사용하고, 이 배열에 대한 포인터와 배열의 길이를 리스트 헤드 구조체에 유지합니다.

이것은 리스트 인덱싱 `a[i]`의 비용이 리스트의 크기나 인덱스의 값과 무관한 연산으로 만듭니다.

항목이 추가되거나 삽입되면, 참조 배열의 크기가 조정됩니다. 항목을 반복적으로 추가하는 성능을 향상하기 위해 약간 영리하게 처리합니다; 배열을 확장해야 할 때, 추가 공간이 할당되어 다음 몇 번에는 실제 크기 조정이 필요하지 않습니다.

3.19 CPython에서 딕셔너리는 어떻게 구현되니까?

CPython의 딕셔너리는 크기 조정 가능한 해시 테이블로 구현됩니다. B-트리와 비교해, 대부분의 상황에서 조회(지금까지 가장 흔한 연산) 성능이 향상되고, 구현이 더 간단합니다.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key and a per-process seed; for example, 'Python' could hash to -539294296 while 'python', a string that differs by a single bit, could hash to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time - $O(1)$, in Big-O notation - to retrieve a key.

3.20 딕셔너리 키가 불변이어야 하는 이유는 무엇입니까?

딕셔너리의 해시 테이블 구현은 키값에서 계산된 해시값을 사용하여 키를 찾습니다. 키가 가변 객체이면, 값이 변경될 수 있어서, 해시도 변경될 수 있습니다. 그러나 키 객체를 변경하는 주체는 그것이 딕셔너리 키로 사용되고 있음을 알 수 없기 때문에, 딕셔너리에서 항목을 이동할 수 없습니다. 그런 다음, 딕셔너리에서 같은 객체를 찾으려고 하면 해시값이 다르기 때문에 찾을 수 없습니다. 이전 값을 찾으려고 해도 해당 해시 저장소에서 발견된 객체의 값이 다르기 때문에 역시 찾을 수 없습니다.

딕셔너리를 리스트로 인덱싱하려면, 먼저 리스트를 튜플로 변환하십시오; `tuple(L)` 함수는 리스트 `L`과 같은 항목을 가진 튜플을 만듭니다. 튜플은 불변이므로 딕셔너리 키로 사용할 수 있습니다.

제안되었지만 받아들여지지 않은 몇 가지 해법:

- 주소(객체 ID)로 리스트를 해시 합니다. 같은 값으로 새 리스트를 생성하면 찾을 수 없기 때문에 작동하지 않습니다; 예를 들어:

```
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

는 두 번째 줄에 사용된 `[1, 2]`의 id가 첫 번째 줄의 것과 다르기 때문에 `KeyError` 예외가 발생합니다. 즉, 딕셔너리 키는 `is`가 아니라 `==`를 사용하여 비교해야 합니다.

- 리스트를 키로 사용할 때 사본을 만듭니다. 가변 객체인 리스트가 자신에 대한 참조를 포함할 수 있고, 복사 코드가 무한 루프에 빠질 수 있기 때문에 작동하지 않습니다.
- 리스트를 키로 허용하지만, 사용자에게 수정하지 않도록 지시합니다. 이것은 잊거나 실수로 리스트를 수정했을 때 프로그램에 추적하기 어려운 버그를 만듭니다. 또한 딕셔너리의 중요한 불변성을 깨뜨립니다; `d.keys()`의 모든 값은 딕셔너리의 키로 사용할 수 있다.
- 딕셔너리 키로 사용되면 리스트를 읽기 전용으로 표시합니다. 문제는 그 값을 변경할 수 있는 것은 최상위 객체만이 아니라는 것입니다; 리스트를 포함하는 튜플을 키로 사용할 수 있습니다. 무엇이든 키로 딕셔너리에 입력하면 거기에서 도달할 수 있는 모든 객체를 읽기 전용으로 표시해야 합니다 - 그리고 다시, 자기 참조 객체는 무한 루프를 일으킬 수 있습니다.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

for i, el in enumerate(l):
    try:
        result = result + (hash(el) % 9999999) * 1001 + i
    except Exception:
        result = (result % 7777777) + i * 333
return result

```

해시 계산은 리스트의 일부 멤버가 해시 불가능할 가능성과 산술 오버플로의 가능성으로 인해 복잡함에 유의하십시오.

또한 객체가 딕셔너리에 있는지에 관계없이, 항상 `o1 == o2` (즉 `o1.__eq__(o2) is True`) 이면 `hash(o1) == hash(o2)` (즉 `o1.__hash__() == o2.__hash__()`) 여야 합니다. 이러한 제한 사항을 충족하지 못하면 딕셔너리와 다른 해시 기반 구조가 오작동합니다.

In the case of `ListWrapper`, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.

3.21 `list.sort()`가 정렬된 리스트를 반환하지 않는 이유는 무엇입니까?

성능이 중요한 상황에서, 단지 정렬하기 위해 리스트를 복사하는 것은 낭비입니다. 따라서, `list.sort()`는 리스트를 제자리에서 정렬합니다. 이 사실을 상기시키기 위해, 정렬된 리스트를 반환하지 않습니다. 이렇게 하면, 정렬된 복사본이 필요하지만 정렬되지 않은 버전을 유지해야 할 때 실수로 리스트를 덮어쓰지 않도록 합니다.

새 리스트를 반환하려면, 대신 내장 `sorted()` 함수를 사용하십시오. 이 함수는 제공된 이터러블에서 새 리스트를 만들고, 정렬한 다음 반환합니다. 예를 들어, 정렬된 순서로 딕셔너리의 키를 이터레이트 하는 방법은 다음과 같습니다:

```

for key in sorted(mydict):
    ... # do whatever with mydict[key]...

```

3.22 파이썬에서 인터페이스 명세를 어떻게 지정하고 강제합니까?

C++와 Java와 같은 언어에서 제공하는 모듈에 대한 인터페이스 명세는 모듈의 메서드와 함수에 대한 프로토타입을 설명합니다. 많은 사람은 컴파일 타임에 인터페이스 명세를 적용하는 것이 대규모 프로그램을 구축하는 데 도움이 된다고 생각합니다.

파이썬 2.6은 추상 베이스 클래스(ABC)를 정의할 수 있는 `abc` 모듈을 추가합니다. 그런 다음 `isinstance()`와 `issubclass()`를 사용하여 인스턴스나 클래스가 특정 ABC를 구현하는지 확인할 수 있습니다. `collections.abc` 모듈은 `Iterable`, `Container` 및 `MutableMapping`과 같은 유용한 ABC 집합을 정의합니다.

파이썬의 경우, 구성 요소에 대한 적절한 테스트 규율을 통해 인터페이스 명세의 많은 이점을 얻을 수 있습니다.

모듈에 대한 좋은 테스트 스위트는 회귀 테스트를 제공함과 동시에 모듈 인터페이스 명세와 예제 집합으로 사용할 수 있습니다. 많은 파이썬 모듈은 스크립트로 실행하여 간단한 “자체 테스트”를 제공할 수 있습니다. 복잡한 외부 인터페이스를 사용하는 모듈조차도 외부 인터페이스의 간단한 “스텝(stub)” 에뮬레이션을 사용하여 종종 격리 테스트 할 수 있습니다. `doctest`와 `unittest` 모듈 또는 제삼자 테스트 프레임워크를 사용하여 모듈의 모든 코드 줄을 실행하는 포괄적인 테스트 스위트를 구축할 수 있습니다.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `list.append()` method is expected to add new elements to the end of some internal list; an interface specification cannot test that your `list.append()` implementation will actually do this correctly, but it's trivial to check this property in a test suite.

테스트 스위트를 작성하는 것은 매우 도움이 되며, 쉽게 테스트 할 수 있도록 코드를 설계할 수 있습니다. 점점 더 많이 사용되는 기술인 테스트 기반 개발(test-driven development)에서는, 실제 코드를 작성하기 전에 먼저 테스트 스위트의 일부를 작성해야 합니다. 물론 파이썬은 여러분이 지지분해지거나 테스트 케이스를 전혀 작성하지 않을 수 있도록 허락합니다.

3.23 goto가 없는 이유는 무엇입니까?

In the 1970s people realized that unrestricted goto could lead to messy “spaghetti” code that was hard to understand and revise. In a high-level language, it is also unneeded as long as there are ways to branch (in Python, with `if` statements and `or`, `and`, and `if/else` expressions) and loop (with `while` and `for` statements, possibly containing `continue` and `break`).

One can also use exceptions to provide a “structured goto” that works even across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the `go` or `goto` constructs of C, Fortran, and other languages. For example:

```
class label(Exception): pass # declare a label

try:
    ...
    if condition: raise label() # goto label
    ...
except label: # where to goto
    pass
...
```

This doesn't allow you to jump into the middle of a loop, but that's usually considered an abuse of `goto` anyway. Use sparingly.

3.24 날 문자열(r-strings)이 역 슬래시로 끝날 수 없는 이유는 무엇입니까?

더 정확하게는, 홀수 개의 역 슬래시로 끝날 수 없습니다: 끝의 쌍이 없는 역 슬래시는 닫는 따옴표 문자를 이스케이프 하여, 끝나지 않은 문자열을 남깁니다.

날 문자열은 자체 역 슬래시 이스케이프 처리를 수행하려는 프로세서(주로 정규식 엔진)에 대한 입력을 쉽게 만들 수 있도록 설계되었습니다. 이러한 프로세서는 일치하지 않는 후행 역 슬래시를 에러로 간주하므로, 날 문자열은 이를 허용하지 않습니다. 그 대가로, 역 슬래시로 이스케이프 하여 문자열 따옴표 문자를 전달할 수 있습니다. 이 규칙은 r-문자열이 의도된 목적으로 사용될 때 잘 작동합니다.

윈도우 경로명을 빌드하려는 경우, 모든 윈도우 시스템 호출은 슬래시도 허용함에 유의하십시오:

```
f = open("/mydir/file.txt") # works fine!
```

DOS 명령에 대한 경로명을 빌드하려는 경우, 예를 들어, 다음 중 하나를 시도하십시오

```
dir = r"\\this\\is\\my\\dos\\dir" "\\\"
dir = r"\\this\\is\\my\\dos\\dir\" "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

3.25 왜 파이썬에는 어트리뷰트 대입을 위한 “with” 문이 없습니까?

Python has a `with` statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some languages have a construct that looks like this:

```
with obj:
    a = 1          # equivalent to obj.a = 1
    total = total + 1  # obj.total = obj.total + 1
```

파이썬에서는, 이러한 구조가 모호해집니다.

오브젝트 파스칼, 델파이 및 C++와 같은 다른 언어는 정적 형을 사용하므로, 어떤 멤버가 대입되고 있는지 명확하게 알 수 있습니다. 이것이 정적 타이핑의 요점입니다 - 컴파일러는 항상 컴파일 시점에 모든 변수의 스코프를 알고 있습니다.

파이썬은 동적 형을 사용합니다. 실행 시간에 어떤 어트리뷰트가 참조되는지 미리 알 수 없습니다. 멤버 어트리뷰트는 실행 중에 객체에 추가하거나 제거할 수 있습니다. 이로 인해 단순한 읽기만으로는 어떤 어트리뷰트가 참조되는 중인지 알 수 없습니다: 지역 변수, 전역 변수 또는 멤버 어트리뷰트?

예를 들어, 다음과 같은 불완전한 스니펫을 보십시오:

```
def foo(a):
    with a:
        print(x)
```

The snippet assumes that `a` must have a member attribute called `x`. However, there is nothing in Python that tells the interpreter this. What should happen if `a` is, let us say, an integer? If there is a global variable named `x`, will it be used inside the `with` block? As you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of `with` and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

이렇게 작성하십시오:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

이름 연결은 파이썬에서 실행 시간에 결정되는데, 두 번째 버전은 확인을 한 번만 수행하면 되므로 실행 속도를 높이는 부작용도 있습니다.

Similar proposals that would introduce syntax to further reduce code volume, such as using a 'leading dot', have been rejected in favour of explicitness (see <https://mail.python.org/pipermail/python-ideas/2016-May/040070.html>).

3.26 Why don't generators support the with statement?

For technical reasons, a generator used directly as a context manager would not work correctly. When, as is most common, a generator is used as an iterator run to completion, no closing is needed. When it is, wrap it as `contextlib.closing(generator)` in the `with` statement.

3.27 if/while/def/class 문에 콜론이 필요한 이유는 무엇입니까?

콜론은 주로 가독성을 높이기 위해 필요합니다 (실험적 ABC 언어의 결과 중 하나입니다). 이것을 고려해 보십시오:

```
if a == b:
    print(a)
```

와

```
if a == b:
    print(a)
```

두 번째 것이 어떻게 약간 더 읽기 쉬운지 주목하십시오. 이 FAQ 답변에서 콜론이 어떻게 예제를 시작하는지도 주목하십시오; 영어의 표준 사용법입니다.

또 다른 사소한 이유는 콜론이 구문 강조 표시가 있는 편집기를 도와준다는 것입니다: 들여쓰기를 늘려야 하는 때를 결정하기 위해, 프로그램 텍스트를 더 정교하게 구문 분석하는 대신 콜론을 찾을 수 있습니다.

3.28 파이썬은 왜 리스트와 튜플 끝에 쉼표를 허용합니까?

파이썬은 리스트, 튜플 및 딕셔너리 끝에 후행 쉼표를 추가할 수 있도록 합니다:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7], # last trailing comma is optional but good style
}
```

이를 허용하는 데에는 몇 가지 이유가 있습니다.

리스트, 튜플 또는 딕셔너리에 대한 리터럴 값이 여러 줄에 걸쳐 있는 경우, 이전 줄에 쉼표를 추가할 필요가 없기 때문에 요소를 추가하기가 더 쉽습니다. 문법 에러를 만들지 않고 줄을 재정렬할 수도 있습니다.

실수로 쉼표를 누락하면 진단하기 어려운 에러가 발생할 수 있습니다. 예를 들면:

```
x = [
    "fee",
    "fie"
    "foo",
    "fum"
]
```

이 리스트에는 네 개의 요소가 있는 것처럼 보이지만, 실제로는 세 가지 요소만 있습니다: “fee”, “fiefoo” 및 “fum”. 항상 쉼표를 추가하면 이러한 에러 원인을 피할 수 있습니다.

후행 쉼표를 허용하면 프로그래밍적인 코드 생성이 더 쉬워질 수도 있습니다.

라이브러리와 확장 FAQ

4.1 일반 라이브러리 관련 질문

4.1.1 작업 X를 수행할 모듈이나 응용 프로그램을 어떻게 찾습니까?

관련 표준 라이브러리 모듈이 있는지 라이브러리 레퍼런스를 확인하십시오. (결국에는 표준 라이브러리에 있는 내용을 배우고 이 단계를 건너뛸 수 있게 됩니다.)

For third-party packages, search the [Python Package Index](#) or try [Google](#) or another web search engine. Searching for “Python” plus a keyword or two for your topic of interest will usually find something helpful.

4.1.2 math.py (socket.py, regex.py 등) 소스 파일은 어디에 있습니까?

모듈의 소스 파일을 찾을 수 없으면 C, C++ 또는 기타 컴파일된 언어로 구현된 내장이나 동적으로 로드된 모듈일 수 있습니다. 이 경우 소스 파일이 없거나 C 소스 디렉터리(파이썬 경로에 없는)의 `mathmodule.c`와 같은 파일일 수 있습니다.

파이썬에는 (적어도) 세 가지 종류의 모듈이 있습니다:

- 1) 파이썬으로 작성된 모듈 (.py);
- 2) C로 작성되고 동적으로 로드되는 모듈 (.dll, .pyd, .so, .sl 등);
- 3) C로 작성되고 인터프리터와 링크된 모듈; 이 목록을 얻으려면, 다음을 입력하십시오:

```
import sys
print(sys.builtin_module_names)
```

4.1.3 유닉스에서 파이썬 스크립트를 실행 파일로 만들려면 어떻게 해야 합니까?

두 가지를 해야 합니다: 스크립트 파일의 모드는 실행 가능해야 하고 첫 번째 줄은 `#!`로 시작하고 그 뒤에 파이썬 인터프리터 경로가 있어야 합니다.

첫 번째는 `chmod +x scriptfile`이나 아마도 `chmod 755 scriptfile`을 실행하여 수행됩니다.

두 번째는 여러 가지 방법으로 수행 할 수 있습니다. 가장 간단한 방법은 다음과 같은 줄을

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

파이썬 인터프리터가 플랫폼에 설치된 경로 이름을 사용하여 파일의 첫 번째 줄로 작성하는 것입니다.

스크립트가 파이썬 인터프리터가 있는 위치와 독립적으로 되도록 하려면, `env` 프로그램을 사용할 수 있습니다. 파이썬 인터프리터가 사용자 `PATH`의 디렉터리에 있다고 가정하면, 거의 모든 유닉스 변형이 다음을 지원합니다:

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

이것을 CGI 스크립트에 적용하지 마십시오. CGI 스크립트의 `PATH` 변수는 종종 최소한이라서, 인터프리터의 실제 절대 경로명을 사용해야 합니다.

때때로, 사용자 환경이 가득 차서 `/usr/bin/env` 프로그램이 실패합니다; 또는 `env` 프로그램이 아예 없습니다. 이 경우, 다음과 같은 핵을 시도할 수 있습니다 (Alex Rezinsky의 기법입니다):

```
#!/bin/sh
""" : """
exec python $0 ${1+"$@"}
"""
```

작은 단점은 이것이 스크립트의 `__doc__` 문자열을 정의한다는 것입니다. 그러나, 다음을 추가하여 문제를 해결할 수 있습니다

```
__doc__ = """...Whatever..."""
```

4.1.4 파이썬 용 `curses/termcap` 패키지가 있습니까?

유닉스 변형의 경우: 표준 파이썬 소스 배포판은 기본적으로 컴파일되지는 않지만, `Modules` 서브 디렉터리에 `curses` 모듈을 포함합니다. (윈도우 배포판에서는 사용할 수 없습니다 - 윈도우용 `curses` 모듈은 없습니다.)

`curses` 모듈은 기본 `curses` 기능뿐만 아니라 색상, 대체 문자 집합 지원, 패드 및 마우스 지원과 같은 `ncurses`와 `SVSV curses`의 많은 추가 기능을 지원합니다. 이는 모듈이 BSD `curses`만 있는 운영 체제와 호환되지 않음을 뜻하지만, 현재 유지 보수되는 OS들은 어느 것도 이 범주에 속하지 않는 것 같습니다.

4.1.5 파이썬에 C의 `onexit()`와 동등한 것이 있습니까?

The `atexit` module provides a register function that is similar to C's `onexit()`.

4.1.6 시그널 처리기가 작동하지 않는 이유는 무엇입니까?

가장 흔한 문제점은 시그널 처리기가 잘못된 인자 목록으로 선언되는 것입니다. 이렇게 호출됩니다

```
handler(signum, frame)
```

따라서 두 개의 매개 변수로 선언해야 합니다:

```
def handler(signum, frame):
    ...
```

4.2 일반적인 작업

4.2.1 파이썬 프로그램이나 컴포넌트를 어떻게 테스트합니까?

파이썬에는 두 가지 테스트 프레임워크가 있습니다. `doctest` 모듈은 모듈의 독스트링에 있는 예제를 찾고 실행한 후, 출력을 독스트링에 제공된 예상 출력과 비교합니다.

`unittest` 모듈은 Java와 Smalltalk 테스트 프레임 워크에서 모델링 된 더 멋진 테스트 프레임워크입니다.

테스트를 더 쉽게 하려면, 여러분의 프로그램에 좋은 모듈식 디자인을 사용해야 합니다. 프로그램은 거의 모든 기능을 함수나 클래스 메서드로 캡슐화해야 합니다 - 그리고 이는 때로 프로그램을 더 빠르게 실행하

는 놀라운 효과가 있습니다 (지역 변수 액세스가 전역 액세스보다 빠르기 때문에). 또한 프로그램은 전역 변수를 변경하는 것에 의존하지 않아야 합니다, 이렇게 하면 테스트하기가 훨씬 어렵기 때문입니다.

프로그램의 “전역 메인 논리”는 다음과 같은 코드를

```
if __name__ == "__main__":
    main_logic()
```

프로그램의 메인 모듈 하단에 넣는 것처럼 간단할 수 있습니다.

일단 프로그램이 다루기 쉬운 함수와 클래스 동작의 모음으로 구성되면, 이 동작을 검사하는 테스트 함수를 작성해야 합니다. 일련의 테스트를 자동화하는 테스트 스위트는 각 모듈과 연관될 수 있습니다. 이것은 많은 작업처럼 들리지만, 파이썬이 아주 간결하고 유연하기 때문에 놀랍도록 쉽습니다. “프로덕션 코드”와 함께 테스트 함수를 작성하여 코딩을 훨씬 더 즐겁고 재미있게 만들 수 있습니다. 버그를 쉽게 찾고 결함을 조기에 발견할 수 있기 때문입니다.

프로그램의 메인 모듈이 아닌 “지원 모듈”에는 모듈의 자체 테스트가 포함될 수 있습니다.

```
if __name__ == "__main__":
    self_test()
```

복잡한 외부 인터페이스와 상호 작용하는 프로그램조차도 파이썬으로 구현된 “가짜” 인터페이스를 사용하여 외부 인터페이스를 사용할 수 없을 때 테스트할 수 있습니다.

4.2.2 독스트링으로 설명서를 어떻게 만듭니까?

The `pydoc` module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is `epydoc`. `Sphinx` can also include docstring content.

4.2.3 한 번에 하나의 키 입력을 받는 방법은 무엇입니까?

유닉스 변형에는 몇 가지 해결책이 있습니다. `curses`를 사용하여 이 작업을 수행하는 것은 간단하지만, `curses`는 배우기에 상당히 큰 모듈입니다.

4.3 스레드

4.3.1 스레드를 사용하여 어떻게 프로그래밍합니까?

`_thread` 모듈이 아닌 `threading` 모듈을 사용하십시오. `threading` 모듈은 `_thread` 모듈이 제공하는 저수준 프리미티브 위에 편리한 추상화를 구축합니다.

4.3.2 제 스레드가 아무것도 실행되지 않는 것 같습니다: 왜 그런가요?

메인 스레드가 종료되자마자, 모든 스레드가 죽습니다. 메인 스레드가 너무 빨리 실행되어, 스레드가 작업을 수행할 시간이 없습니다.

간단한 수정은 프로그램 끝에 모든 스레드가 완료될 만큼 충분히 긴 휴면을 추가하는 것입니다:

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10) # <-----!>
```

그러나 이제 (많은 플랫폼에서) 스레드는 병렬로 실행되지 않고, 한 번에 하나씩 순차적으로 실행되는 것처럼 보입니다! 그 이유는 OS 스레드 스케줄러가 이전 스레드가 블록 될 때까지 새 스레드를 시작하지 않기 때문입니다.

간단한 수정은 실행 함수의 시작 부분에 작은 휴면을 추가하는 것입니다:

```
def thread_task(name, n):
    time.sleep(0.001) # <-----!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

`time.sleep()` 을 위한 좋은 지연 값을 추측하는 대신, 일종의 세마포어 메커니즘을 사용하는 것이 좋습니다. 한 가지 아이디어는 `queue` 모듈을 사용하여 큐 객체를 만들고, 각 스레드가 완료될 때 큐에 토큰을 추가하게 하고, 메인 스레드가 스레드 수 만큼의 토큰을 읽도록 하는 것입니다.

4.3.3 여러 작업자 스레드 간에 작업을 어떻게 배달합니까?

가장 쉬운 방법은 `concurrent.futures` 모듈, 특히 `ThreadPoolExecutor` 클래스를 사용하는 것입니다.

또는, 디스패치 알고리즘을 세밀하게 제어하려면, 직접 논리를 작성할 수 있습니다. `queue` 모듈을 사용하여 작업 목록을 포함하는 큐를 만드십시오. `Queue` 클래스는 객체 목록을 유지하고 큐에 항목을 추가하는 `.put(obj)` 메서드와 이를 반환하는 `.get()` 메서드를 갖습니다. 클래스는 각 작업이 정확히 한 번만 전달 되도록 하는 데 필요한 lock을 관리합니다.

간단한 예를 들면 다음과 같습니다:

```
import threading, queue, time

# The worker thread gets jobs off the queue. When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.current_thread(), end=' ')
            print('queue empty')
            break
        else:
            print('Worker', threading.current_thread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

실행하면 다음과 같은 출력이 생성됩니다:

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

자세한 내용은 모듈 설명서를 참조하십시오. Queue 클래스는 기능이 풍부한 인터페이스를 제공합니다.

4.3.4 어떤 종류의 전역 값 변경이 스레드 안전합니까?

내부적으로 전역 인터프리터 록(GIL)이 사용되어 한 번에 하나의 스레드 만 파이썬 VM에서 실행되도록 합니다. 일반적으로, 파이썬은 바이트 코드 명령어들 사이에서만 스레드 간 전환을 제공합니다; `sys.setswitchinterval()` 을 통해 얼마나 자주 전환할지를 설정할 수 있습니다. 따라서 각 바이트 코드 명령어와 각 명령어에서 도달하는 모든 C 구현 코드는 파이썬 프로그램의 관점에서 원자적입니다.

이론적으로, 이것은 정확하게 따지기 위해서는 PVM 바이트 코드 구현에 대한 정확한 이해가 필요하다는 것을 의미합니다. 경험적으로는, 이것은 “원자 적으로 보이는” 내장 데이터형(정수, 리스트, 딕셔너리 등)의 공유 변수에 대한 조작이 실제로 원자 적임을 의미합니다.

예를 들어, 다음 연산은 모두 원자 적입니다 (L, L1, L2는 리스트, D, D1, D2는 딕셔너리, x, y는 객체, i, j는 정수입니다):

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

이것들은 아닙니다:

```
i = i+1
L.append(L[-1])
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
L[i] = L[j]
D[x] = D[x] + 1
```

Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

4.3.5 전역 인터프리터 록을 제거할 수 없습니까?

전역 인터프리터 록(GIL)은 종종 하이 엔드 다중 프로세서 서버 기계에 파이썬을 배치하는 데 방해가 된다고 여겨집니다, (거의) 모든 파이썬 코드가 GIL을 잡고 있는 동안에만 실행하려고 해서, 다중 스레드 파이썬 프로그램이 사실상 오직 하나의 CPU만 사용하기 때문입니다.

With the approval of [PEP 703](#) work is now underway to remove the GIL from the CPython implementation of Python. Initially it will be implemented as an optional compiler flag when building the interpreter, and so separate builds will be available with and without the GIL. Long-term, the hope is to settle on a single build, once the performance implications of removing the GIL are fully understood. Python 3.13 is likely to be the first release containing this work, although it may not be completely functional in this release.

The current work to remove the GIL is based on a [fork of Python 3.9 with the GIL removed](#) by Sam Gross. Prior to that, in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the “free threading” patches) that removed the GIL and replaced it with fine-grained locking. Adam Olsen did a similar experiment in his [python-safethread](#) project. Unfortunately, both of these earlier experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL. The Python 3.9 fork is the first attempt at removing the GIL with an acceptable performance impact.

The presence of the GIL in current Python releases doesn't mean that you can't make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. The `ProcessPoolExecutor` class in the new `concurrent.futures` module provides an easy way of doing so; the `multiprocessing` module provides a lower-level API in case you want more control over dispatching of tasks.

C 확장을 신중하게 사용하는 것도 도움이 됩니다; C 확장을 사용하여 시간이 오래 걸리는 작업을 수행하면, 확장은 실행 스레드가 C 코드에 있는 동안 GIL을 해제하고 다른 스레드가 어떤 작업을 수행할 수 있도록 할 수 있습니다. `zlib`와 `hashlib`와 같은 일부 표준 라이브러리 모듈은 이미 이렇게 합니다.

An alternative approach to reducing the impact of the GIL is to make the GIL a per-interpreter-state lock rather than truly global. This was first implemented in Python 3.12 and is available in the C API. A Python interface to it is expected in Python 3.13. The main limitation to it at the moment is likely to be 3rd party extension modules, since these must be written with multiple interpreters in mind in order to be usable, so many older extension modules will not be usable.

4.4 입력과 출력

4.4.1 파일을 어떻게 삭제합니까? (그리고 다른 파일 질문들...)

`os.remove(filename)` 이나 `os.unlink(filename)` 을 사용하십시오; 설명서는 `os` 모듈을 참조하십시오. 두 함수는 동일합니다; `unlink()` 는 단순히 이 함수에 대한 유닉스 시스템 호출의 이름입니다.

디렉토리를 제거하려면, `os.rmdir()` 을 사용하십시오; 만들려면 `os.mkdir()` 을 사용하십시오. `os.makedirs(path)` 는 존재하지 않는 `path`의 중간 디렉토리를 만듭니다. `os.removedirs(path)` 는 비어 있는 한, 중간 디렉토리를 제거합니다; 전체 디렉터리 트리과 그 내용을 삭제하려면 `shutil.rmtree()` 를 사용하십시오.

파일 이름을 바꾸려면 `os.rename(old_path, new_path)` 를 사용하십시오.

파일을 자르려면, `f = open(filename, "rb+")` 를 사용하여 열고 `f.truncate(offset)` 을 사용하십시오; `offset`의 기본값은 현재 탐색(`seek`) 위치입니다. `os.open()` 으로 열린 파일의 경우 `os.ftruncate(fd, offset)` 도 있습니다. 여기서 `fd`는 파일 기술자(작은 정수)입니다.

shutil 모듈에도 `copyfile()`, `copytree()` 및 `rmtree()` 를 포함한 파일에서 작동하는 많은 함수가 포함되어 있습니다.

4.4.2 파일을 어떻게 복사합니까?

The `shutil` module contains a `copyfile()` function. Note that on Windows NTFS volumes, it does not copy `alternate data streams` nor `resource forks` on macOS HFS+ volumes, though both are now rarely used. It also doesn't copy file permissions and metadata, though using `shutil.copy2()` instead will preserve most (though not all) of it.

4.4.3 바이너리 데이터를 읽는 (또는 쓰는) 방법은 무엇입니까?

복잡한 바이너리 데이터 형식을 읽거나 쓰려면, `struct` 모듈을 사용하는 것이 가장 좋습니다. 바이너리 데이터(보통 숫자)를 포함하는 문자열을 취해서 파이썬 객체로 변환할 수 있도록 합니다; 그리고 그 반대도 가능합니다.

예를 들어, 다음 코드는 파일에서 빅 엔디안 형식의 두 개의 2-바이트 정수와 하나의 4-바이트 정수를 읽습니다:

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

포맷 문자열의 '>' 는 빅 엔디안 데이터를 강제합니다; 문자 'h' 는 하나의 “짧은(short) 정수”(2바이트)를 읽고, 'l' 은 문자열에서 하나의 “긴(long) 정수”(4바이트)를 읽습니다.

더욱 규칙적인 데이터(예를 들어 `int` 나 `float` 의 동종 리스트)의 경우, `array` 모듈을 사용할 수도 있습니다.

참고

바이너리 데이터를 읽고 쓰려면, 바이너리 모드로 파일을 열어야 합니다 (여기서는, "rb"를 `open()` 으로 전달합니다). 대신 "r"(기본값)을 사용하면, 파일이 텍스트 모드로 열리고 `f.read()` 는 `bytes` 객체 대신 `str` 객체를 반환합니다.

4.4.4 `os.popen()`으로 만든 파이프에서 `os.read()`를 사용할 수 없는 것처럼 보입니다; 왜입니까?

`os.read()` 는 열린 파일을 나타내는 작은 정수인 파일 기술자를 취하는 저수준 함수입니다. `os.popen()` 은 내장 `open()` 함수에서 반환하는 것과 같은 형의 고수준 파일 객체를 만듭니다. 따라서, `os.popen()` 으로 만들어진 파이프 `p`에서 `n` 바이트를 읽으려면, `p.read(n)` 을 사용해야 합니다.

4.4.5 직렬 (RS232) 포트에 어떻게 액세스합니까?

For Win32, OSX, Linux, BSD, Jython, IronPython:

`pyserial`

유닉스의 경우, Mitch Chapman의 유즈넷 게시물을 참조하십시오:

<https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com>

4.4.6 왜 `sys.stdout(stdin, stderr)`을 닫아도 닫히지 않습니까?

파이썬 파일 객체는 저수준 C 파일 기술자의 고수준 추상화 계층입니다.

내장 `open()` 함수를 통해 파이썬에서 만드는 대부분 파일 객체의 경우, `f.close()` 는 파이썬 파일 객체를 파이썬의 관점에서 닫은 것으로 표시하고 하부 C 파일 기술자를 닫도록 합니다. 이것은 `f` 가 가비지가 될 때 `f` 의 파괴자에서 자동으로 일어나기도 합니다.

그러나 `stdin`, `stdout` 및 `stderr`은 파이썬에서 특별하게 처리되는데, C 역시 이들에게 특수한 상태를 부여하기 때문입니다. `sys.stdout.close()` 를 실행하면 파이썬 수준 파일 객체가 닫힌 것으로 표시되지만, 연관된 C 파일 기술자를 닫지 않습니다.

이 세 가지 중 하나에 대한 하부 C 파일 기술자를 닫으려면, 먼저 이것이 정말로 여러분이 하고 싶은 것인지 확인해야 합니다 (예를 들어, I/O를 수행하려는 확장 모듈이 혼동할 수 있습니다). 그렇다면, `os.close()` 를 사용하십시오:

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

또는 숫자 상수 0, 1 및 2를 각각 사용할 수 있습니다.

4.5 네트워크/인터넷 프로그래밍

4.5.1 파이썬에는 어떤 WWW 도구가 있습니까?

라이브러리 레퍼런스 매뉴얼의 `internet`과 `netdata` 장을 참조하십시오. 파이썬에는 서버 측과 클라이언트 측 웹 시스템을 구축하는 데 도움이 되는 많은 모듈이 있습니다.

사용 가능한 프레임워크 요약은 Paul Boddie 가 <https://wiki.python.org/moin/WebProgramming>에서 유지 관리합니다.

4.5.2 HTML 생성을 위해 어떤 모듈을 사용해야 합니까?

`Web Programming wiki page`에서 유용한 링크 모음을 찾을 수 있습니다.

4.5.3 파이썬 스크립트에서 메일을 보내려면 어떻게 해야 합니까?

표준 라이브러리 모듈 `smtplib`를 사용하십시오.

다음은 이를 사용하는 매우 간단한 대화식 메일 발신기입니다. 이 방법은 SMTP 리스너를 지원하는 모든 호스트에서 작동합니다.

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

유닉스 전용 대안은 `sendmail`을 사용합니다. `sendmail` 프로그램의 위치는 시스템마다 다릅니다; 때로는 `/usr/lib/sendmail`, 때로는 `/usr/sbin/sendmail`. `sendmail` 매뉴얼 페이지가 도움이 될 것입니다. 샘플 코드는 다음과 같습니다:

```
import os

SENDMAIL = "/usr/sbin/sendmail" # sendmail location
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n") # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

4.5.4 소켓의 `connect()` 메서드에서 블로킹을 피하려면 어떻게 해야 할까요?

`select` 모듈이 소켓의 비동기 I/O를 지원하는 데 흔히 사용됩니다.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno.errno.EINPROGRESS` indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the `errno` value. To poll, you can call `connect_ex()` again later - `0` or `errno.EISCONN` indicate that you're connected - or you can pass this socket to `select.select()` to check if it's writable.

참고

The `asyncio` module provides a general purpose single-threaded and concurrent asynchronous library, which can be used for writing non-blocking network code. The third-party `Twisted` library is a popular and feature-rich alternative.

4.6 데이터베이스

4.6.1 파이썬에 데이터베이스 패키지에 대한 인터페이스가 있습니까?

예.

DBM과 GDBM 같은 디스크 기반 해시에 대한 인터페이스도 표준 파이썬에 포함되어 있습니다. 경량 디스크 기반 관계형 데이터베이스를 제공하는 `sqlite3` 모듈도 있습니다.

대부분 관계형 데이터베이스에 대한 지원이 제공됩니다. 자세한 내용은 [DatabaseProgramming wiki page](#)를 참조하십시오.

4.6.2 파이썬에서 영속 객체를 어떻게 구현할까요?

`pickle` 라이브러리 모듈은 이것을 매우 일반적인 방식으로 해결하고 (여전히 열린 파일, 소켓 또는 창과 같은 것을 저장할 수는 없지만), `shelve` 라이브러리 모듈은 `pickle`과 `(g)dbm`을 사용하여 임의의 파이썬 객체를 포함하는 영속적(persistent) 매핑을 만듭니다.

4.7 수학과 숫자

4.7.1 파이썬에서 난수를 어떻게 생성할까요?

표준 모듈 `random`은 난수 생성기를 구현합니다. 사용법은 간단합니다:

```
import random
random.random()
```

This returns a random floating-point number in the range [0, 1).

이 모듈에는 다른 많은 특수 생성기가 있습니다, 가령:

- `randrange(a, b)` 는 `[a, b)` 범위의 정수를 선택합니다.
- `uniform(a, b)` chooses a floating-point number in the range `[a, b)`.
- `normalvariate(mean, sdev)` 는 정규 (가우시안) 분포를 샘플링합니다.

일부 고수준 함수는 시퀀스에서 직접 작동합니다, 가령:

- `choice(S)` 는 주어진 시퀀스에서 무작위 요소 하나를 선택합니다.
- `shuffle(L)` 은 리스트를 제자리에서 섞습니다, 즉 무작위로 순서를 바꿉니다.

독립적인 여러 개의 난수 생성기를 만들기 위해 인스턴스화 할 수 있는 `Random` 클래스도 있습니다.

5.1 C로 나만의 함수를 만들 수 있습니까?

그렇습니다. 함수, 변수, 예외 및 심지어 새로운 형을 포함하는 내장 모듈을 C로 만들 수 있습니다. `extending-index` 문서에 설명되어 있습니다.

대부분의 중급이나 고급 파이썬 서적에서도 이 주제를 다룰 것입니다.

5.2 C++로 나만의 함수를 만들 수 있습니까?

그렇습니다, C++에 있는 C 호환성 기능을 사용합니다. 파이썬 인클루드(include) 파일 주위에 `extern "C"` { ... }를 배치하고 파이썬 인터프리터가 호출할 각 함수 앞에 `extern "C"`를 배치하십시오. 생성자를 가진 전역이나 정적(static) C++ 객체는 대개 좋은 생각이 아닙니다.

5.3 C를 쓰는 것은 어렵습니다; 대안이 있습니까?

수행하려는 작업에 따라, 여러분 만의 C 확장을 작성하는 여러 가지 대안이 있습니다.

`Cython` and its relative `Pyrex` are compilers that accept a slightly modified form of Python and generate the corresponding C code. `Cython` and `Pyrex` make it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as `SWIG`. `SIP`, `CXX Boost`, or `Weave` are also alternatives for wrapping C++ libraries.

5.4 C에서 임의의 파이썬 문장을 어떻게 실행할 수 있습니까?

이를 수행하는 최상위 수준 함수는 `PyRun_SimpleString()`이며, 이는 모듈 `__main__`의 컨텍스트에서 실행될 단일 문자열 인자를 취하고 성공하면 0을 반환하고 (`SyntaxError`를 포함하는) 예외가 발생하면 -1을 반환합니다. 더 많은 제어를 원하면, `PyRun_String()`을 사용하십시오; `Python/pythonrun.c`에 있는 `PyRun_SimpleString()` 소스를 참조하십시오.

5.5 C에서 임의의 파이썬 표현식을 어떻게 평가할 수 있습니까?

이전 질문에서 나온 `PyRun_String()` 함수를 `start` 기호 `Py_eval_input` 을 사용하여 호출하십시오; 표현식을 구문 분석하고, 평가하고 값을 반환합니다.

5.6 파이썬 객체에서 C값을 어떻게 추출합니까?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C's `strlen()` should not be used.

객체의 형을 검사하려면, 먼저 `NULL` 이 아닌지 확인한 다음 `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()` 등을 사용하십시오.

소위 '추상' 인터페이스가 제공하는 파이썬 객체에 대한 고수준 API도 있습니다 - 자세한 내용은 `Include/abstract.h` 를 읽으십시오. `PySequence_Length()`, `PySequence_GetItem()` 등과 같은 호출로 모든 종류의 파이썬 시퀀스와 인터페이스 할 수 있을 뿐만 아니라 숫자(`PyNumber_Index()` 등)와 `PyMapping` API의 매핑과 같은 다른 많은 유용한 프로토콜을 지원합니다.

5.7 `Py_BuildValue()`를 사용하여 임의 길이의 튜플을 만드는 방법은 무엇입니까?

그릴 수 없습니다. 대신 `PyTuple_Pack()` 을 사용하십시오.

5.8 C에서 객체의 메서드를 어떻게 호출합니까?

`PyObject_CallMethod()` 함수는 객체의 임의의 메서드를 호출하는 데 사용할 수 있습니다. 매개 변수는 객체, 호출할 메서드의 이름, `Py_BuildValue()` 에 사용되는 것과 같은 포맷 문자열 및 인자 값입니다:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

메서드가 있는 모든 객체에서 작동합니다 - 내장이나 사용자 정의 모두 작동합니다. 반환 값을 `Py_DECREF()` 할 책임은 여러분에게 있습니다.

예를 들어, 인자 10, 0으로 파일 객체의 "seek" 메서드를 호출하려면 (파일 객체 포인터가 "f" 라고 가정합니다):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
    ... an exception occurred ...
}
else {
    Py_DECREF(res);
}
```

`PyObject_CallObject()` 는 항상 인자 목록에 대한 튜플을 원하므로, 인자 없이 함수를 호출하려면, format 으로 "()" 를 전달하고, 하나의 인자로 함수를 호출하려면, 인자를 괄호로 묶습니다, 예를 들어 "(i)".

5.9 PyErr_Print()의 출력(또는 stdout/stderr로 인쇄되는 모든 것)을 어떻게 잡습니까?

파이썬 코드에서, `write()` 메서드를 지원하는 객체를 정의하십시오. 이 객체를 `sys.stdout`과 `sys.stderr`에 대입하십시오. `print_error`를 호출하거나 표준 트레이스백 메커니즘이 작동하도록 두십시오. 그러면 출력은 여러분의 `write()` 메서드가 보내는 곳으로 갑니다.

이렇게 하는 가장 쉬운 방법은 `io.StringIO` 클래스를 사용하는 것입니다:

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

같은 작업을 수행하는 사용자 정의 객체는 다음과 같습니다:

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!
```

5.10 C에서 파이썬으로 작성된 모듈에 어떻게 액세스합니까?

다음과 같이 모듈 객체에 대한 포인터를 얻을 수 있습니다:

```
module = PyImport_ImportModule("<modulename>");
```

모듈을 아직 임포트하지 않았으면 (즉, `sys.modules`에 아직 없으면), 이것은 모듈을 초기화합니다; 그렇지 않으면 단순히 `sys.modules["<modulename>"]`의 값을 반환합니다. 이것은 모듈을 어떤 이름 공간에도 넣지 않음에 유의하십시오 - 단지 초기화되도록 하고 `sys.modules`에 저장되도록 합니다.

그런 다음, 다음과 같이 모듈의 어트리뷰트(즉 모듈에 정의된 모든 이름)에 액세스할 수 있습니다:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

모듈에 있는 변수에 대입하기 위해 `PyObject_SetAttrString()`을 호출하는 것도 작동합니다.

5.11 파이썬에서 C++ 객체에 어떻게 인터페이스 합니까?

요구 사항에 따라 여러 가지 접근 방식이 있습니다. 이 작업을 수동으로 수행하려면, “확장 및 내장” 문서를 읽는 것으로 시작하십시오. 파이썬 런타임 시스템의 경우 C와 C++ 사이에는 큰 차이가 없다는 것을 상기하십시오 - 따라서 C 구조체 (포인터) 형을 중심으로 새로운 파이썬 형을 작성하는 전략이 C++ 객체에도 적용됩니다.

C++ 라이브러리의 경우, C를 쓰는 것은 어렵습니다; 대안이 있습니까?를 참조하십시오.

5.12 Setup 파일을 사용하여 모듈을 추가했는데 make가 실패합니다; 왜 그럴습니까?

Setup은 개행으로 끝나야 하며, 개행이 없으면 빌드 프로세스가 실패합니다. (이 문제를 해결하려면 지저분한 셸 스크립트 해킹이 필요하며, 이 버그는 너무 사소해서 그런 노력을 들일만 한 가치가 없는 것 같습니다.)

5.13 확장을 어떻게 디버깅합니까?

동적으로 로드된 확장에 GDB를 사용할 때, 확장이 로드될 때까지 확장에 중단점을 설정할 수 없습니다.

.gdbinit 파일에서 (또는 대화식으로) 다음 명령을 추가하십시오:

```
br _PyImport_LoadDynamicModule
```

그런 다음, GDB를 실행할 때:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 리눅스 시스템에서 파이썬 모듈을 컴파일하고 싶지만, 일부 파일이 없습니다. 왜 그럴습니까?

Most packaged versions of Python omit some files required for compiling Python extensions.

For Red Hat, install the python3-devel RPM to get the necessary files.

For Debian, run `apt-get install python3-dev`.

5.15 “잘못된 입력”과 “불완전한 입력”을 어떻게 구별할 수 있습니까?

때로 파이썬 대화식 인터프리터의 동작을 흉내 내고 싶을 때가 있습니다. 이것은 입력이 불완전할 때 (예를 들어, “if” 문의 시작을 입력했거나 괄호나 삼중 문자열 따옴표를 닫지 않았을 때) 계속 프롬프트를 표시하지만, 입력이 유효하지 않으면 즉시 문법 에러 메시지를 표시합니다.

파이썬에서는 codeop 모듈을 사용할 수 있습니다. 이 모듈은 구문 분석기의 동작을 충분히 근사합니다. 예를 들어, IDLE은 이것을 사용합니다.

C에서 이렇게 하는 가장 쉬운 방법은 `PyRun_InteractiveLoop()` 를 호출하고 (아마 별도의 스레드에서), 파이썬 인터프리터가 입력을 처리하도록 하는 것입니다. `PyOS_ReadlineFunctionPointer()` 가 여러분의 사용자 정의 입력 함수를 가리키도록 설정할 수도 있습니다. 자세한 힌트는 `Modules/readline.c`와 `Parser/myreadline.c`를 참조하십시오.

5.16 정의되지 않은 g++ 기호 __builtin_new나 __pure_virtual을 어떻게 찾을 수 있습니까?

g++ 확장 모듈을 동적으로 로드하려면, 파이썬을 다시 컴파일하고, g++를 사용하여 다시 링크하고 (파이썬 `Modules/Makefile`에서 `LINKCC`를 변경하십시오), g++를 사용하여 여러분의 확장 모듈을 링크해야 합니다 (예를 들어, `g++ -shared -o mymodule.so mymodule.o`).

5.17 일부 메서드는 C로 구현되고 그 밖의 것은 파이썬으로 구현된 (예를 들어 상속을 통해) 객체 클래스를 만들 수 있습니까?

그렇습니다, `int`, `list`, `dict` 등과 같은 내장 클래스를 상속할 수 있습니다.

The Boost Python Library (BPL, <https://www.boost.org/libs/python/doc/index.html>) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).

윈도우 파이썬 FAQ

6.1 윈도우에서 파이썬 프로그램을 실행하려면 어떻게 해야 하나요?

이 질문은 명확한 질문이 아닙니다. 이미 윈도우 명령 줄에서 프로그램을 실행하는 것에 익숙하다면 이 모든 것이 분명할 것입니다. 그렇지 않으면, 좀 더 지침이 필요할 수 있습니다.

Unless you use some sort of integrated development environment, you will end up *typing* Windows commands into what is referred to as a “Command prompt window”. Usually you can create such a window from your search bar by searching for `cmd`. You should be able to recognize when you have started such a window because you will see a Windows “command prompt”, which usually looks like this:

```
C:\>
```

아마 글자가 다르고, 그 뒤에 다른 것들이 있을 수 있습니다. 그러므로 다음과 같은 것을 쉽게 볼 수 있습니다:

```
D:\YourName\Projects\Python>
```

컴퓨터 설정 방법과 최근에 끝낸 그 밖의 것에 의존적입니다. 일단 이런 창을 시작했다면, 파이썬 프로그램을 실행할 준비가 된 것입니다.

파이썬 스크립트는 파이썬 인터프리터 라는 다른 프로그램에서 진행할 필요가 있다는 것을 알고 있어야 합니다. 인터프리터는 스크립트를 읽고, 바이트 코드로 컴파일하고, 바이트 코드를 실행하여 프로그램을 구동합니다. 그렇다면, 인터프리터를 통해 파이썬을 처리하려면 어떻게 준비해야 하겠습니까?

먼저, 명령 창이 “py”라는 단어를 해석기를 시작하는 지시로 인식하는지 확인해야 합니다. 명령 창을 열고 있으면, 명령 `py` 를 입력하고 `return` 키를 눌러야 합니다:

```
C:\Users\YourName> py
```

다음과 같이 표시됩니다:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
> on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

인터프리터를 “대화형 모드”로 시작했습니다. 즉, 파이썬 문장이나 표현식을 대화식으로 입력하여 대기하는 중에 실행 혹은 평가시킬 수 있습니다. 이것은 파이썬의 가장 강력한 기능 중 하나입니다. 선택한 몇

가지 수식을 입력하여 확인하고 그 결과를 확인하십시오:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

많은 사람이 대화형 모드를 편리하면서도 프로그래밍이 가능한 계산기로 사용합니다. 대화형 파이썬 세션을 종료하려면, `exit()` 함수를 호출하거나 `z` 를 입력하면서 `Ctrl` 키를 누르고 나서 “Enter” 키를 눌러 윈도우 명령 프롬프트로 돌아갑니다.

또한, 시작 ▶ 프로그램 ▶ *Python 3.x* ▶ *Python*(명령 줄) 메뉴 선택과 같은 시작 메뉴 항목을 찾아볼 수 있으며, 그 결과 새 창에서 `>>>` 프롬프트를 볼 수 있습니다. 이 경우, `exit()` 함수를 호출하거나 `Ctrl-Z` 를 입력하면 창이 사라집니다. 윈도우에서 단일 “python” 명령을 실행하는 중이고, 인터프리터를 종료하면 창을 닫습니다.

이제 `py` 명령이 인식되었으므로, 당신은 당신의 파이썬 스크립트를 줄 수 있습니다. 파이썬 스크립트에 절대 경로나 상대 경로를 지정해야 할 것입니다. 당신의 파이썬 스크립트가 `hello.py` 라는 이름으로 당신의 데스크톱에 위치한다고 하면, 명령 프롬프트가 홈 디렉토리에 잘 열려있으므로 다음과 같이 내용이 표시됩니다:

```
C:\Users\YourName>
```

그래서 이제 스크립트 경로 뒤에 `py` 를 입력하여 파이썬 스크립트를 제공하기 위해 `py` 명령을 요청할 것입니다:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

6.2 파이썬 스크립트 실행 파일로 만들려면 어떻게 해야 합니까?

윈도우에서 표준 파이썬 설치관리자는 이미 `.py` 확장자와 파일 유형(`Python.File`)을 연결했고, 인터프리터(`D:\Program Files\Python\python.exe "%1" %*`)를 실행하는 `open` 명령의 파일 유형을 제공합니다. 이것은 명령 프롬프트에서 ‘`foo.py`’ 과 같은 스크립트 실행파일을 만들기에 충분합니다. 확장자명 없이 ‘`foo`’ 를 입력하여 스크립트를 실행하려면 `PATHEXT` 환경 변수에 `.py`를 추가해야 합니다.

6.3 왜 때때로 파이썬은 시작하는 데 시간이 오래 걸립니까?

일반적으로 파이썬은 윈도우에서 매우 빠르게 시작되지만, 때때로 파이썬을 시작하는 데 갑자기 오랜 시간이 걸린다는 버그 보고서가 있습니다. 이러한 것은 파이썬이 동일하게 구성된 것으로 보이는 다른 윈도우 시스템에서 잘 작동하기 때문에 더욱 더 곤혹스럽게 만듭니다.

이 문제는 해당 컴퓨터의 바이러스 검사 소프트웨어의 잘못된 설정으로 발생하는 것일 수 있습니다. 일부 바이러스 스캐너는 파일 시스템으로부터 읽은 모든 것을 모니터링하도록 스캐너를 구성할 때 두 자릿수 규모의 시동 오버헤드를 도입하는 것으로 알려져 있습니다. 시스템에서 바이러스 검사 소프트웨어의 구성을 확인하여 실제로 동일하게 구성되었는지 확인하십시오. 모든 파일 시스템 읽기 작업을 검색하도록 구성된 경우, McAfee는 특히 문제를 일으킵니다.

6.4 파이썬 스크립트에서 실행 파일을 만드는 방법은 무엇입니까?

See [파이썬 스크립트로 독립 실행형 바이너리를 만들려면 어떻게 해야 합니까?](#) for a list of tools that can be used to make executables.

6.5 *.pyd 파일은 DLL과 동일합니까?

예, .pyd 파일은 dll 이지만, 몇 가지 차이점이 있습니다. 만약 당신이 foo.pyd 라는 이름의 DLL을 가지고 있다면, PyInit_foo() 함수를 반드시 가지고 있어야 합니다. 당신은 파이썬 “import foo”를 쓸 수 있으며, 파이썬은 (foo.py, foo.pyc 뿐만 아니라) foo.pyd 를 검색할 것이고, 이를 발견하면 초기화하기 위해 PyInit_foo() 호출을 시도할 것입니다. 윈도우에서 DLL의 존재를 요구할 것이기 때문에 .exe를 foo.lib와 링크하면 안 됩니다.

foo.pyd에 대한 검색 경로는 윈도우에서 foo.dll을 검색하는 데 사용하는 경로가 아닌 PYTHONPATH임을 유의하십시오. 또한, 프로그램을 실행하기 위해 foo.pyd가 있을 필요는 없지만, 프로그램을 dll과 링크한 경우에는 dll이 필요합니다. 물론, import foo를 하기 위해서는 foo.pyd가 필요합니다. DLL에서 링크지는 소스 코드에서 __declspec(dllexport)로 선언됩니다. .pyd에서 링크지는 사용 가능한 함수 목록에 정의됩니다.

6.6 윈도우 응용프로그램에 파이썬을 포함하려면 어떻게 해야 합니까?

윈도우 앱에서 파이썬 인터프리터를 포함하려면 다음과 같이 요약할 수 있습니다:

1. Do **not** build Python into your .exe file directly. On Windows, Python must be a DLL to handle importing modules that are themselves DLL's. (This is the first key undocumented fact.) Instead, link to pythonNN.dll; it is typically installed in C:\Windows\System. NN is the Python version, a number such as “33” for Python 3.3.

두 가지 방법으로 파이썬에 링크할 수 있습니다. 로드 타임 링크는 pythonNN.lib에 대한 링크를 의미하고, 런타임 링크는 pythonNN.dll에 대한 링크를 의미합니다. (일반 참고: pythonNN.lib는 pythonNN.dll에 해당하는 소위 “import lib”입니다. 오직 링커를 위해 기호만 정의합니다)

런타임 링크는 링크 옵션을 크게 단순화합니다. 모든 것은 런타임 중에 발생합니다. 당신의 코드는 윈도우 LoadLibraryEx() 루틴을 사용하여 pythonNN.dll을 로드해야 합니다. 그 코드는 윈도우 GetProcAddress() 루틴에서 얻은 포인터를 사용하여 pythonNN.dll (파이썬의 C API)의 액세스 루틴 및 데이터를 사용해야 합니다. 매크로는 이러한 포인터를 파이썬 C API에서 루틴을 호출하는 모든 C 코드에 투명하게 사용할 수 있습니다.

2. If you use SWIG, it is easy to create a Python “extension module” that will make the app's data and methods available to Python. SWIG will handle just about all the grungy details for you. The result is C code that you link *into* your .exe file (!) You do **not** have to create a DLL file, and this also simplifies linking.
3. SWIG는 확장 모듈의 이름에 따라 이름이 달라지는 초기화 함수(C 함수)를 생성합니다. 예를 들어, 모듈의 이름이 leo인 경우, 초기화 함수를 initleo()로 합니다. SWIG 새도 클래스를 사용하면 initleo()로 합니다. 이것은 새도 클래스가 사용하는 대부분 숨겨진 조력자 클래스를 초기화합니다.

2단계에서 C 코드를 .exe 파일에 링크할 수 있는 이유는 초기화 함수를 호출하는 것이 모듈을 파이썬으로 임포트하는 것과 동일하기 때문입니다! (이것이 문서화되지 않은 두 번째 사실입니다)

4. 간단히 말해서, 당신은 당신의 확장 모듈로 파이썬 인터프리터를 초기화하기 위해 다음 코드를 사용할 수 있습니다.

```
#include <Python.h>
...
Py_Initialize(); // Initialize Python.
initmyApp(); // Initialize (import) the helper class.
PyRun_SimpleString("import myApp"); // Import the shadow class.
```

5. 파이썬의 C API에는 두 가지 문제가 있는데, 이것은 당신이 PythonNN.dll을 만드는 데 사용되는 컴파일러인 MSVC 이외의 컴파일러를 사용하면 명백해집니다.

Problem 1: The so-called “Very High Level” functions that take FILE * arguments will not work in a multi-compiler environment because each compiler's notion of a struct FILE will be different. From an implementation standpoint these are very low level functions.

문제 2: 반환값이 void인 C 함수의 래퍼를 생성할 때 SWIG는 다음과 같은 코드를 생성합니다:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

아아, `Py_None`은 `pythonNN.dll` 내부의 `_Py_NoneStruct`라는 복잡한 데이터 구조에 대한 참조로 확장하는 매크로입니다. 다시 말하자면, 이 코드는 멀티 컴파일러 환경에서 실패할 것입니다. 다음과 같은 코드로 바꾸십시오:

```
return Py_BuildValue("");
```

저는 이것을 작동시키지는 못했지만, `SWIG`의 `%typemap` 명령을 사용하여 자동으로 변경하는 것이 가능할지도 모릅니다(저는 완전 `SWIG` 초보자입니다).

- 윈도우 앱 내부에서 파이썬 셸 스크립트를 사용하여 파이썬 인터프리터 창을 설치하는 것은 좋은 생각이 아닙니다. 결과 창은 앱의 창 시스템과는 독립적인 것입니다. 오히려, 당신(또는 `wxPythonWindow` 클래스)은 “native” 인터프리터 창을 만들어야 합니다. 이 창은 파이썬 인터프리터와 연결하기 쉽습니다. 당신은 파이썬의 `i/o`를 읽기 및 쓰기를 지원하는 모든 객체로 리디렉션할 수 있으므로 `read()`와 `write()` 메서드를 포함하는 파이썬 객체(확장 모듈에 정의됨)만 있으면 됩니다.

6.7 편집기가 내 파이썬 소스에 탭을 삽입하지 않도록 하려면 어떻게 해야 하나요?

FAQ는 탭을 사용하는 것을 권장하지 않으며, 파이썬 스타일 안내서([PEP 8](#))는 분산된 파이썬 코드에 대해 4 개의 스페이스를 권장합니다. 이 또한 Emacs `python-mode`의 기본값입니다.

모든 편집기에서 탭과 스페이스를 혼용하는 것은 좋은 생각이 아닙니다. 이 점에서 `MSVC`는 다르지 않으며 스페이스를 사용하기 쉽게 구성됩니다: 다음 행동을 따라 해보십시오 `Tools ▸ Options ▸ Tabs`, 그리고 파일 유형은 “Default”로 하고 “Tab size”와 “Indent size”는 4로 설정하고, “Insert spaces”를 라디오 버튼으로 선택합니다.

만일 혼용된 탭과 스페이스로 인해 선행 공백에 문제가 발생하는 경우, 파이썬이 `IndentationError`나 `TabError`를 발생시킵니다. 또한, `tabnanny` 모듈을 실행하여 배치 모드에서 디렉터리 트리를 확인할 수도 있습니다.

6.8 블로킹 없이 키 입력을 확인하려면 어떻게 해야 하나요?

`msvcrt` 모듈을 사용합니다. 이것은 표준 윈도우-특정 확장 모듈입니다. 이것은 키보드 히트가 존재하는지를 확인하는 `kbhit()`와 에코 없이 문자를 얻는 `getch()`를 정의합니다.

6.9 How do I solve the missing api-ms-win-crt-runtime-l1-1-0.dll error?

This can occur on Python 3.5 and later when using Windows 8.1 or earlier without all updates having been installed. First ensure your operating system is supported and is up to date, and if that does not resolve the issue, visit the [Microsoft support page](#) for guidance on manually installing the C Runtime update.

그래픽 사용자 인터페이스 FAQ

7.1 일반적인 GUI 질문

7.2 What GUI toolkits exist for Python?

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called tkinter. This is probably the easiest to install (since it comes included with most [binary distributions](#) of Python) and use. For more info about Tk, including pointers to the source, see the [Tcl/Tk home page](#). Tcl/Tk is fully portable to the macOS, Windows, and Unix platforms.

Depending on what platform(s) you are aiming at, there are also several alternatives. A [list of cross-platform and platform-specific](#) GUI frameworks can be found on the python wiki.

7.3 Tkinter 질문

7.3.1 Tkinter 응용 프로그램을 어떻게 고정(freeze)합니까?

Freeze는 독립 실행형 응용 프로그램을 만드는 도구입니다. Tkinter 응용 프로그램을 고정할 때, 응용 프로그램은 여전히 Tcl과 Tk 라이브러리가 필요하므로 진정한 독립 실행형이 아닙니다.

One solution is to ship the application with the Tcl and Tk libraries, and point to them at run-time using the `TCL_LIBRARY` and `TK_LIBRARY` environment variables.

Various third-party freeze libraries such as py2exe and cx_Freeze have handling for Tkinter applications built-in.

7.3.2 I/O를 기다리는 동안 Tk 이벤트를 처리하도록 할 수 있습니까?

On platforms other than Windows, yes, and you don't even need threads! But you'll have to restructure your I/O code a bit. Tk has the equivalent of Xt's `XtAddInput()` call, which allows you to register a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. See [tkinter-file-handlers](#).

7.3.3 Tkinter에서 키 바인딩이 동작하지 않습니다: 이유가 무엇입니까?

An often-heard complaint is that event handlers bound to events with the `bind()` method don't get handled even when the appropriate key is pressed.

가장 흔한 원인은 바인딩이 적용되는 위젯에 “키보드 포커스”가 없는 것입니다. `focus` 명령에 대한 Tk 설명서를 확인하십시오. 보통 위젯은 그것을 클릭할 때 키보드 포커스를 받습니다 (레이블은 그렇지 않습니다; `takefocus` 옵션을 보십시오).

“왜 내 컴퓨터에 파이썬이 설치되어 있습니까?” FAQ

8.1 파이썬이 무엇입니까?

파이썬은 프로그래밍 언어입니다. 많은 다른 응용 프로그램에 사용됩니다. 파이썬은 배우기 쉬우므로 일부 고등학교와 대학에서는 입문 프로그래밍 언어로 사용되지만, Google, NASA 및 Lucasfilm Ltd.와 같은 곳에서 전문 소프트웨어 개발자가 사용하기도 합니다.

파이썬에 대해 더 알고 싶다면, [파이썬 입문자 지침서](#)부터 시작하십시오.

8.2 내 컴퓨터에 파이썬이 설치된 이유는 무엇입니까?

파이썬이 시스템에 설치되어 있지만 설치한 기억이 없다면, 그렇게 될 수 있는 몇 가지 가능한 방법이 있습니다.

- 아마도 컴퓨터의 다른 사용자가 프로그래밍을 배우고 싶어 하고 그것을 설치했을 것입니다; 당신은 누가 컴퓨터를 사용했고 설치했는지 추측해야 합니다.
- 컴퓨터에 설치된 제삼자 응용 프로그램이 파이썬으로 작성되었으며, 파이썬 설치를 포함할 수 있습니다. GUI 프로그램에서 네트워크 서버와 관리 스크립트에 이르기까지 그런 응용 프로그램이 많이 있습니다.
- 일부 윈도우 컴퓨터에는 파이썬이 설치되어 있습니다. 이 글을 쓰는 시점에, 우리는 파이썬이 포함된 Hewlett-Packard와 Compaq의 컴퓨터에 대해 알고 있습니다. 분명히 HP/Compaq의 관리 도구 중 일부가 파이썬으로 작성되었을 겁니다.
- 맥 OS 및 일부 리눅스 배포판과 같은 많은 유닉스 호환 운영 체제에는 기본적으로 파이썬이 설치되어 있습니다; 기본 설치에 포함되어 있습니다.

8.3 파이썬을 삭제할 수 있습니까?

파이썬이 어디서 왔는지에 달려 있습니다.

누군가 의도적으로 설치했다면, 아무 문제도 일으키지 않고 제거할 수 있습니다. 윈도우에서는, 제어판의 프로그램 추가/제거 아이콘을 사용하십시오.

제삼자 응용 프로그램에서 파이썬을 설치했다면, 제거할 수도 있지만, 해당 응용 프로그램이 더는 작동하지 않게 됩니다. 파이썬을 직접 제거하는 대신 해당 응용 프로그램의 제거 프로그램을 사용해야 합니다.

파이썬이 운영체제와 함께 제공되었다면, 제거하는 것은 바람직하지 않습니다. 제거하면, 파이썬으로 작성된 모든 도구가 더는 실행되지 않으며, 그중 일부는 중요할 수 있습니다. 문제를 해결하려면 다시 시스템을 재설치해야 할 수 있습니다.

>>>

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

...

다음과 같은 것들을 가리킬 수 있습니다:

- 들여쓰기 된 코드 블록의 코드를 입력할 때, 쌍을 이루는 구분자 (괄호, 대괄호, 중괄호) 안에 코드를 입력할 때, 데코레이터 지정 후의 대화형 셸의 기본 파이썬 프롬프트.
- Ellipsis 내장 상수.

abstract base class (추상 베이스 클래스)

추상 베이스 클래스는 `hasattr()` 같은 다른 테크닉들이 불편하거나 미묘하게 잘못된 (예를 들어, 매직 메서드) 경우, 인터페이스를 정의하는 방법을 제공함으로써 **덱 타이핑** 을 보완합니다. ABC는 가상 서브 클래스를 도입하는데, 클래스를 계승하지 않으면서도 `isinstance()` 와 `issubclass()` 에 의해 감지될 수 있는 클래스들입니다; abc 모듈 설명서를 보세요. 파이썬에는 많은 내장 ABC들이 따라오는데 다음과 같은 것들이 있습니다: 자료 구조 (`collections.abc` 모듈에서), 숫자 (`numbers` 모듈에서), 스트림 (`io` 모듈에서), 임포트 파인더와 로더 (`importlib.abc` 모듈에서). abc 모듈을 사용해서 자신만의 ABC를 만들 수도 있습니다.

annotate function

A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

annotation (어노테이션)

관습에 따라 **형 힌트** 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 반환 값과 연결된 레이블입니다.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see *annotations-howto* for best practices on working with annotations.

argument (인자)

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

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

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

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

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

인자는 함수 바디의 이름 붙은 지역 변수에 대입됩니다. 이 대입에 적용되는 규칙들에 대해서는 [calls](#) 절을 보세요. 문법적으로, 어떤 표현식이건 인자로 사용될 수 있습니다; 구해진 값이 지역 변수에 대입됩니다.

용어집의 [매개변수](#) 항목과 FAQ 질문 [인자와 매개변수의 차이](#) 와 [PEP 362](#)도 보세요.

asynchronous context manager (비동기 컨텍스트 관리자)

`__aenter__()` 와 `__aexit__()` 메서드를 정의함으로써 `async with` 문에서 보이는 환경을 제어하는 객체. [PEP 492](#)로 도입되었습니다.

asynchronous generator (비동기 제너레이터)

비동기 제너레이터 *이터레이터* 를 돌려주는 함수. `async def` 로 정의되는 코루틴 함수처럼 보이는데, `async for` 루프가 사용할 수 있는 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다.

보통 비동기 제너레이터 함수를 가리키지만, 어떤 문맥에서는 비동기 제너레이터 *이터레이터* 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

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

asynchronous generator iterator (비동기 제너레이터 이터레이터)

비동기 제너레이터 함수가 만드는 객체.

비동기 *이터레이터* 인데 `__anext__()` 를 호출하면 어웨이터블 객체를 돌려주고, 이것은 다음 `yield` 표현식 까지 비동기 제너레이터 함수의 바디를 실행합니다.

각 `yield`는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 `try`-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 *이터레이터* 가 `__anext__()` 가 돌려주는 또 하나의 어웨이터블로 재개되면, 떠난 곳으로 복귀합니다. [PEP 492](#)와 [PEP 525](#)를 보세요.

asynchronous iterable (비동기 이터러블)

`async for` 문에서 사용될 수 있는 객체. `__aiter__()` 메서드는 비동기 *이터레이터* 를 돌려줘야 합니다. [PEP 492](#)로 도입되었습니다.

asynchronous iterator (비동기 이터레이터)

`__aiter__()` 와 `__anext__()` 메서드를 구현하는 객체. `__anext__()` 는 어웨이터블 객체를 돌려줘야 합니다. `async for`는 `StopAsyncIteration` 예외가 발생할 때까지 비동기 *이터레이터*의 `__anext__()` 메서드가 돌려주는 어웨이터블을 푹니다. [PEP 492](#)로 도입되었습니다.

attached thread state

A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds

of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

attribute (어트리뷰트)

흔히 점표현식을 사용하는 이름으로 참조되는 객체와 결합한 값. 예를 들어, 객체 *o*가 어트리뷰트 *a*를 가지면, *o.a*처럼 참조됩니다.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

awaitable (어웨이터블)

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

BDFL

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

binary file (바이너리 파일)

바이트열류 객체들을 읽고 쓸 수 있는 **파일 객체**. 바이너리 파일의 예로는 바이너리 모드 ('rb', 'wb' 또는 'rb+') 로 열린 파일, `sys.stdin.buffer`, `sys.stdout.buffer`, `io.BytesIO` 와 `gzip.GzipFile`의 인스턴스를 들 수 있습니다.

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

borrowed reference (빌린 참조)

In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object (바이트열류 객체)

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

어떤 연산들은 바이너리 데이터가 가변적일 필요가 있습니다. 이런 경우에 설명서는 종종 “읽고-쓰기 바이트열류 객체”라고 표현합니다. 가변 버퍼 객체의 예로는 `bytearray`와 `bytearray`의 `memoryview`가 있습니다. 다른 연산들은 바이너리 데이터가 불변 객체 (“읽기 전용 바이트열류 객체”)에 저장되도록 요구합니다; 이런 것들의 예로는 `bytes`와 `bytes` 객체의 `memoryview`가 있습니다.

bytecode (바이트 코드)

파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현입니다. 바이트 코드는 `.pyc` 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만듭니다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있습니다). 이 “중간 언어”는 각 바이트 코드에 대응하는 기계를 실행하는 **가상 기계**에서 실행된다고 말합니다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 합니다.

바이트 코드 명령어들의 목록은 `dis` 모듈 설명서에 나옵니다.

callable (콜러블)

A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback (콜백)

인자로 전달되는 미래의 어느 시점에서 실행될 서브 루틴 함수.

class (클래스)

사용자 정의 객체들을 만들기 위한 주형. 클래스 정의는 보통 클래스의 인스턴스를 대상으로 연산하는 메서드 정의들을 포함합니다.

class variable (클래스 변수)

클래스에서 정의되고 클래스 수준 (즉, 클래스의 인스턴스에서도 아니라) 에서만 수정되는 변수.

closure variable (클로저 변수)

A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number (복소수)

익숙한 실수 시스템의 확장인데, 모든 숫자가 실수부와 허수부의 합으로 표현됩니다. 허수부는 실수에 허수 단위(-1의 제곱근)를 곱한 것인데, 종종 수학에서는 *i*로, 공학에서는 *j*로 표기합니다. 파이썬은 후자의 표기법을 쓰는 복소수를 기본 지원합니다; 허수부는 *j* 접미사를 붙여서 표기합니다, 예를 들어, `3+1j`. `math` 모듈의 복소수 버전이 필요하다면, `cmath`를 사용합니다. 복소수의 활용은 꽤 수준 높은 수학적 기능입니다. 필요하다고 느끼지 못한다면, 거의 확실히 무시해도 좋습니다.

context (컨텍스트)

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol (컨텍스트 관리 프로토콜)

The `__enter__()` and `__exit__()` methods called by the `with` statement. See [PEP 343](#).

context manager (컨텍스트 관리자)

컨텍스트 관리 프로토콜을 구현하고 `with` 문에서 보이는 환경을 제어하는 객체. [PEP 343](#)을 참조하십시오.

context variable (컨텍스트 변수)

A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous (연속)

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

coroutine (코루틴)

코루틴은 서브루틴의 더 일반화된 형태입니다. 서브루틴은 한 지점에서 진입하고 다른 지점에서 탈출합니다. 코루틴은 여러 다른 지점에서 진입하고, 탈출하고, 재개할 수 있습니다. 이것들은 `async def` 문으로 구현할 수 있습니다. **PEP 492**를 보세요.

coroutine function (코루틴 함수)

코루틴 객체를 돌려주는 함수. 코루틴 함수는 `async def` 문으로 정의될 수 있고, `await` 와 `async for` 와 `async with` 키워드를 포함할 수 있습니다. 이것들은 **PEP 492**에 의해 도입되었습니다.

CPython

파이썬 프로그래밍 언어의 규범적인 구현인데, python.org에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 “CPython” 이 사용됩니다.

current context (현재 컨텍스트)

The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

decorator (데코레이터)

다른 함수를 돌려주는 함수인데, 보통 `@wrapper` 문법을 사용한 함수 변환으로 적용됩니다. 데코레이터의 흔한 예는 `classmethod()` 과 `staticmethod()` 입니다.

데코레이터 문법은 단지 편의 문법일 뿐입니다. 다음 두 함수 정의는 의미상으로 동등합니다:

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

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

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의 와 클래스 정의 의 설명서를 보면 됩니다.

descriptor (디스크립터)

메서드 `__get__()` 이나 `__set__()` 이나 `__delete__()` 를 정의하는 객체. 클래스 어트리뷰트가 디스크립터일 때, 어트리뷰트 조회는 특별한 연결 작용을 일으킵니다. 보통, *a.b*를 읽거나, 쓰거나, 삭제하는데 사용할 때, *a*의 클래스 디렉터리에서 *b* 라고 이름 붙여진 객체를 찾습니다. 하지만 *b*가 디스크립터면, 해당하는 디스크립터 메서드가 호출됩니다. 디스크립터를 이해하는 것은 파이썬에 대한 깊은 이해의 열쇠인데, 함수, 메서드, 프로퍼티, 클래스 메서드, 스태틱 메서드, 슈퍼 클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 `descriptors`나 디스크립터 사용법 안내서에 나옵니다.

dictionary (딕셔너리)

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

dictionary comprehension (딕셔너리 컴프리헨션)

이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 딕셔너리를 반환하는 간결한 방법. `results = {n: n ** 2 for n in range(10)}`은 값 *n ** 2*에 매핑된 키 *n*을 포함하는 딕셔너리를 생성합니다. comprehensions을 참조하십시오.

dictionary view (딕셔너리 뷰)

`dict.keys()`, `dict.values()`, `dict.items()` 메서드가 돌려주는 객체들을 딕셔너리 뷰라고 부릅니다. 이것들은 딕셔너리 항목들에 대한 동적인 뷰를 제공하는데, 딕셔너리가 변경될 때, 뷰가 이 변화를 반영한다는 뜻입니다. 딕셔너리 뷰를 완전한 리스트로 바꾸려면 `list(dictview)`를 사용하면 됩니다. dict-views를 보세요.

docstring (독스트링)

클래스, 함수, 모듈에서 첫 번째 표현식으로 나타나는 문자열 리터럴. 스위트가 실행될 때는 무시되

지만, 컴파일러에 의해 인지되어 둘러싼 클래스, 함수, 모듈의 `__doc__` 어트리뷰트로 삽입됩니다. 인트로스펙션을 통해 사용할 수 있으므로, 객체의 설명서를 위한 규범적인 장소입니다.

duck-typing (덕 타이핑)

올바른 인터페이스를 가졌는지 판단하는데 객체의 형을 보지 않는 프로그래밍 스타일; 대신, 단순히 메서드나 어트리뷰트가 호출되거나 사용됩니다 (“오리처럼 보이고 오리처럼 꺽꺽댄다면, 그것은 오리다.”) 특정한 형 대신에 인터페이스를 강조함으로써, 잘 설계된 코드는 다형적인 치환을 허락함으로써 유연성을 개선할 수 있습니다. 덕 타이핑은 `type()` 이나 `isinstance()` 을 사용한 검사를 피합니다. (하지만, 덕 타이핑이 추상 베이스 클래스로 보완될 수 있음에 유의해야 합니다.) 대신에, `hasattr()` 검사나 *EAFP* 프로그래밍을 씁니다.

EAFP

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

evaluate function

A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression (표현식)

어떤 값으로 구해질 수 있는 문법적인 조각. 다른 말로 표현하면, 표현식은 리터럴, 이름, 어트리뷰트 액세스, 연산자, 함수들과 같은 값을 돌려주는 표현 요소들을 쌓아 올린 것입니다. 다른 많은 언어와 대조적으로, 모든 언어 구성물들이 표현식인 것은 아닙니다. `while`처럼, 표현식으로 사용할 수 없는 문장들이 있습니다. 대입 또한 문장이고, 표현식이 아닙니다.

extension module (확장 모듈)

C 나 C++로 작성된 모듈인데, 파이썬의 C API를 사용해서 핵심이나 사용자 코드와 상호 작용합니다.

f-string (f-문자열)

'f' 나 'F' 를 앞에 붙인 문자열 리터럴들을 흔히 “f-문자열” 이라고 부르는데, 포맷 문자열 리터럴의 줄임말입니다. [PEP 498](#) 을 보세요.

file object (파일 객체)

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

실제로는 세 부류의 파일 객체들이 있습니다. 날(*raw*) *바이너리 파일*, 버퍼드(*buffered*) *바이너리 파일*, 텍스트 파일. 이들의 인터페이스는 `io` 모듈에서 정의됩니다. 파일 객체를 만드는 규범적인 방법은 `open()` 함수를 쓰는 것입니다.

file-like object (파일류 객체)

파일 객체 의 비슷한 말.

filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기)

Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

로케일 인코딩 도 보세요.

finder (파인더)

임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

두 종류의 파인더가 있습니다: `sys.meta_path` 와 함께 사용하는 **메타 경로 파인더** 와 `sys.path_hooks` 과 함께 사용하는 **경로 엔트리 파인더**.

더 자세한 내용은 `finders-and-loaders` 와 `importlib`를 참조하십시오.

floor division (정수 나눗셈)

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

free threading (자유 스레딩)

A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable (자유 변수)

Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function (함수)

호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 **인자**가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. **매개변수** 와 **메서드** 와 `function` 섹션도 보세요.

function annotation (함수 어노테이션)

함수 매개변수나 반환 값의 **어노테이션**.

함수 어노테이션은 일반적으로 **형 힌트**로 사용됩니다: 예를 들어, 이 함수는 두 개의 `int` 인자를 받아들일 것으로 기대되고, 동시에 `int` 반환 값을 줄 것으로 기대됩니다:

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

함수 어노테이션 문법은 `function` 절에서 설명합니다.

이 기능을 설명하는 **변수 어노테이션** 과 **PEP 484**를 참조하세요. 또한 어노테이션에 대한 모범 사례는 `annotations-howto`를 참조하세요.

__future__

A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

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

garbage collection (가비지 수거)

더 사용되지 않는 메모리를 반납하는 절차. 파이썬은 참조 횟수 추적과 참조 순환을 감지하고 끊을 수 있는 순환 가비지 수거기를 통해 가비지 수거를 수행합니다. 가비지 수거기는 `gc` 모듈을 사용해서 제어할 수 있습니다.

generator (제너레이터)

제너레이터 이터레이터를 돌려주는 함수. 일반 함수처럼 보이는데, 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다. 이 값들은 `for`-루프로 사용하거나 `next()` 함수로 한 번에 하나씩 꺼낼 수 있습니다.

보통 제너레이터 함수를 가리키지만, 어떤 문맥에서는 제너레이터 이터레이터를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

generator iterator (제너레이터 이터레이터)

제너레이터 함수가 만드는 객체.

각 `yield`는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 `try`-문들을 포함하는) 실행 상태를 기억합니다. 제네레이터 이터레이터가 재개되면, 떠난 곳으로 복귀합니다 (호출마다 새로 시작하는 함수와 대비됩니다).

generator expression (제네레이터 표현식)

이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 `for` 절과 생략 가능한 `if` 절이 뒤에 붙는 일반 표현식 처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

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

generic function (제네릭 함수)

같은 연산을 서로 다른 형들에 대해 구현한 여러 함수로 구성된 함수. 호출 때 어떤 구현이 사용될지는 디스패치 알고리즘에 의해 결정됩니다.

싱글 디스패치 용어집 항목과 `functools.singledispatch()` 데코레이터와 [PEP 443](#)도 보세요.

generic type (제네릭 형)

매개 변수화 할 수 있는 형; 일반적으로 `list` 나 `dict`와 같은 컨테이너 클래스. 형 힌트와 어노테이션에 사용됩니다.

For more details, see generic alias types, [PEP 483](#), [PEP 484](#), [PEP 585](#), and the `typing` module.

GIL

전역 인터프리터 록 을 보세요.

global interpreter lock (전역 인터프리터 록)

한 번에 오직 하나의 스레드가 파이썬 바이트 코드를 실행하도록 보장하기 위해 CPython 인터프리터가 사용하는 메커니즘. (`dict`와 같은 중요한 내장형들을 포함하는) 객체 모델이 묵시적으로 동시 액세스에 대해 안전하도록 만들어서 CPython 구현을 단순하게 만듭니다. 인터프리터 전체를 잠그는 것은 인터프리터를 다중스레드화하기 쉽게 만드는 대신, 다중 프로세서 기계가 제공하는 병렬성의 많은 부분을 희생합니다.

하지만, 어떤 확장 모듈들은, 표준이나 제삼자 모두, 압축이나 해싱 같은 계산 집약적인 작업을 수행할 때는 GIL을 반납하도록 설계되었습니다. 또한, I/O를 할 때는 항상 GIL을 반납합니다.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see [PEP 703](#).

In prior versions of Python's C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

hash-based pyc (해시 기반 pyc)

유효성을 판별하기 위해 해당 소스 파일의 최종 수정 시간이 아닌 해시를 사용하는 바이트 코드 캐시 파일. `pyc-invalidation`을 참조하세요.

hashable (해시 가능)

객체가 일생 그 값이 변하지 않는 해시값을 갖고 (`__hash__()` 메서드가 필요합니다), 다른 객체와 비교될 수 있으면 (`__eq__()` 메서드가 필요합니다), 해시 가능하다고 합니다. 같다고 비교되는 해시 가능한 객체들의 해시값은 같아야 합니다.

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

대부분 파이썬의 불변 내장 객체들은 해시 가능합니다; (리스트나 딕셔너리 같은) 가변 컨테이너들은 그렇지 않습니다; (튜플이나 `frozenset` 같은) 불변 컨테이너들은 그들의 요소들이 해시 가능할 때만 해시 가능합니다. 사용자 정의 클래스의 인스턴스 객체들은 기본적으로 해시 가능합니다. (자기 자신을 제외하고는) 모두 다르다고 비교되고, 해시값은 `id()`로 부터 만들어집니다.

IDLE

파이썬을 위한 통합 개발 및 학습 환경 (Integrated Development and Learning Environment). `idle`은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

immortal (불멸)

Immortal objects are a CPython implementation detail introduced in [PEP 683](#).

If an object is immortal, its [reference count](#) is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via `PyUnstable_IsImmortal()` in the C API.

immutable (불변)

고정된 값을 갖는 객체. 불변 객체는 숫자, 문자열, 튜플을 포함합니다. 이런 객체들은 변경될 수 없습니다. 새 값을 저장하려면 새 객체를 만들어야 합니다. 변하지 않는 해시값이 있어야 하는 곳에서 중요한 역할을 합니다, 예를 들어, 딕셔너리의 키.

import path (임포트 경로)

[경로 기반 파인더](#)가 임포트 할 모듈을 찾기 위해 검색하는 장소들 (또는 [경로 엔트리](#))의 목록. 임포트 하는 동안, 이 장소들의 목록은 보통 `sys.path`로부터 옵니다, 하지만 서브 패키지의 경우 부모 패키지의 `__path__` 어트리뷰트로부터 올 수도 있습니다.

importing (임포트)

한 모듈의 파이썬 코드가 다른 모듈의 파이썬 코드에서 사용될 수 있도록 하는 절차.

importer (임porter)

모듈을 찾기도 하고 로드 하기도 하는 객체; 동시에 [파인더](#)이자 [로더](#) 객체입니다.

interactive (대화형)

파이썬은 대화형 인터프리터를 갖고 있는데, 인터프리터 프롬프트에서 문장과 표현식을 입력할 수 있고, 즉각 실행된 결과를 볼 수 있다는 뜻입니다. 인자 없이 단지 `python`을 실행하세요 (컴퓨터의 주메뉴에서 선택하는 것도 가능할 수 있습니다). 새 아이디어를 검사하거나 모듈과 패키지를 들여다 보는 매우 강력한 방법입니다 (`help(x)`를 기억하세요). 대화형 모드에 대한 자세한 내용은 [tut-interac](#)를 보세요.

interpreted (인터프리티드)

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

interpreter shutdown (인터프리터 종료)

종료하라는 요청을 받을 때, 파이썬 인터프리터는 특별한 시기에 진입하는데, 모듈이나 여러 가지 중요한 내부 구조들과 같은 모든 할당된 자원들을 단계적으로 반납합니다. 또한, [가비지 수거기](#)를 여러 번 호출합니다. 사용자 정의 파괴자나 `weakref` 콜백에 있는 코드들의 실행을 시작시킬 수 있습니다. 종료 시기 동안 실행되는 코드는 다양한 예외들을 만날 수 있는데, 그것이 의존하는 자원들이 더 기능하지 않을 수 있기 때문입니다 (흔한 예는 라이브러리 모듈이나 경고 장치들입니다).

인터프리터 종료를 주된 원인은 실행되는 `__main__` 모듈이나 스크립트가 실행을 끝내는 것입니다.

iterable (이터러블)

멤버들을 한 번에 하나씩 돌려줄 수 있는 객체. 이터러블의 예로는 모든 (`list`, `str`, `tuple` 같은) 시퀀스 형들, `dict` 같은 몇몇 비 시퀀스 형들, [파일 객체들](#), `__iter__()`나 [시퀀스](#) 개념을 구현하는 `__getitem__()` 메서드를 써서 정의한 모든 클래스의 객체들이 있습니다.

이터러블은 `for` 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 (`zip()`, `map()`, ...)에 사용될 수 있습니다. 이터러블 객체가 내장 함수 `iter()`에 인자로 전달되면, 그 객체의 이터레이터를 돌려줍니다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효합니다. 이터러블을 사용할 때, 보통은 `iter()`를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없습니다. `for` 문은 이것들을 여러분을 대신해서 자동으로 해주는데, 루프를 도는 동안 이터레이터를 잡아들 이름 없는 변수를 만듭니다. [이터레이터](#), [시퀀스](#), [제너레이터](#)도 보세요.

iterator (이터레이터)

데이터의 스트림을 표현하는 객체. 이터레이터의 `__next__()` 메서드를 반복적으로 호출하면 (또는 내장 함수 `next()`로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 `StopIteration` 예외를 일으킵니다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 `__next__()` 메서드 호출은 `StopIteration` 예외를 다시 일으키기만 합니다. 이터레이터는 이터레이터 객체 자신을 돌려주는 `__iter__()` 메서드를 가질 것이 요구되기 때문에, 이터레이터는

이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드입니다. (`list` 같은) 컨테이너 객체는 `iter()` 함수로 전달하거나 `for` 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

`typeiter` 에 더 자세한 내용이 있습니다.

CPython 구현 상세: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function (키 함수)

키 함수 또는 콜레이션(collation) 함수는 정렬(sorting)이나 배열(ordering)에 사용되는 값을 돌려주는 콜러블입니다. 예를 들어, `locale.strxfrm()` 은 로케일 특정 방식을 따르는 정렬 키를 만드는데 사용됩니다.

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

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

keyword argument (키워드 인자)

[인자](#) 를 보세요.

lambda (람다)

호출될 때 값이 구해지는 하나의 표현식으로 구성된 이름 없는 인라인 함수. 람다 함수를 만드는 문법은 `lambda [parameters]: expression` 입니다.

LBYL

뛰기 전에 보라 (Look before you leap). 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 사전 조건들을 검사합니다. 이 스타일은 [EAFP](#) 접근법과 대비되고, 많은 `if` 문의 존재로 특징지어집니다.

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

lexical analyzer (어휘 분석기)

Formal name for the *tokenizer*; see [token](#).

list (리스트)

내장 파이썬 시퀀스. 그 이름에도 불구하고, 원소에 대한 액세스가 $O(1)$ 이기 때문에, 연결 리스트(linked list)보다는 다른 언어의 배열과 유사합니다.

list comprehension (리스트 컴프리헨션)

시퀀스의 요소들 전부 또는 일부를 처리하고 그 결과를 리스트로 돌려주는 간결한 방법. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` 는 0에서 255 사이에 있는 짝수들의 16진수 (0x..) 들을 포함하는 문자열의 리스트를 만듭니다. `if` 절은 생략할 수 있습니다. 생략하면, `range(256)` 에 있는 모든 요소가 처리됩니다.

loader (로더)

An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also:

- [finders-and-loaders](#)
- `importlib.abc.Loader`
- [PEP 302](#)

locale encoding (로케일 인코딩)

On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: "cp1252").

On Android and VxWorks, Python uses "utf-8" as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the [filesystem encoding and error handler](#).

magic method (매직 메서드)

특수 메서드의 비공식적인 비슷한 말.

mapping (매핑)

임의의 키 조회를 지원하고 `collections.abc.Mapping` 이나 `collections.abc.MutableMapping` 추상 베이스 클래스에 지정된 메서드들을 구현하는 컨테이너 객체. 예로는 `dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` 를 들 수 있습니다.

meta path finder (메타 경로 파인더)

`sys.meta_path`의 검색이 돌려주는 [파인더](#). 메타 경로 파인더는 [경로 엔트리 파인더](#)와 관련되어 있기는 하지만 다릅니다.

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

metaclass (메타 클래스)

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

metaclasses에서 더 자세한 내용을 찾을 수 있습니다.

method (메서드)

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

method resolution order (메서드 결정 순서)

메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서입니다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 `python_2.3_mro`를 보세요.

module (모듈)

파이썬 코드의 조직화 단위를 담당하는 객체. 모듈은 임의의 파이썬 객체들을 담은 이름 공간을 갖습니다. 모듈은 [임포트](#) 절차에 의해 파이썬으로 로드됩니다.

[패키지](#)도 보세요.

module spec (모듈 스펙)

모듈을 로드하는데 사용되는 임포트 관련 정보들을 담고 있는 이름 공간. `importlib.machinery.ModuleSpec`의 인스턴스.

`module-specs`도 보세요.

MRO

[메서드 결정 순서](#)를 보세요.

mutable (가변)

가변 객체는 값이 변할 수 있지만 `id()`는 일정하게 유지합니다. [불변](#)도 보세요.

named tuple (네임드 튜플)

“named tuple(네임드 튜플)”이라는 용어는 튜플에서 상속하고 이름 붙은 어트리뷰트를 사용하여 인덱스 할 수 있는 요소에 액세스 할 수 있는 모든 형이나 클래스에 적용됩니다. 형이나 클래스에는 다른 기능도 있을 수 있습니다.

`time.localtime()` 과 `os.stat()` 가 반환한 값을 포함하여, 여러 내장형이 네임드 튜플입니다. 또 다른 예는 `sys.float_info`입니다:

```
>>> sys.float_info[1]                # indexed access
1024
>>> sys.float_info.max_exp           # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

일부 네임드 튜플은 내장형(위의 예)입니다. 또는, `tuple`에서 상속하고 이름 붙은 필드를 정의하는 일반 클래스 정의로 네임드 튜플을 만들 수 있습니다. 이러한 클래스는 직접 작성하거나, `typing.NamedTuple`를 계승하거나 팩토리 함수 `collections.namedtuple()`로 만들 수 있습니다. 후자의 기법은 직접 작성하거나 내장 네임드 튜플에서는 찾을 수 없는 몇 가지 추가 메서드를 추가하기도 합니다.

namespace (이름 공간)

변수가 저장되는 장소. 이름 공간은 딕셔너리로 구현됩니다. 객체에 중첩된 이름 공간(메서드에서) 뿐만 아니라 지역, 전역, 내장 이름 공간이 있습니다. 이름 공간은 이름 충돌을 방지해서 모듈성을 지원합니다. 예를 들어, 함수 `builtins.open`과 `os.open()`은 그들의 이름 공간에 의해 구별됩니다. 또한, 이름 공간은 어떤 모듈이 함수를 구현하는지를 분명하게 만들어서 가독성과 유지 보수성에 도움을 줍니다. 예를 들어, `random.seed()` 또는 `itertools.islice()`라고 쓰면 그 함수들이 각각 `random`과 `itertools` 모듈에 의해 구현되었음이 명확해집니다.

namespace package (이름 공간 패키지)

오직 서브 패키지들의 컨테이너로만 기능하는 패키지. 이름 공간 패키지는 물리적인 실체가 없을 수도 있고, 특히 `__init__.py` 파일이 없으므로 정규 패키지와는 다릅니다.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see [PEP 420](#) and [reference-namespace-package](#).

모듈도 보세요.

nested scope (중첩된 스코프)

둘러싼 정의에서 변수를 참조하는 능력. 예를 들어, 다른 함수 내부에서 정의된 함수는 바깥 함수에 있는 변수들을 참조할 수 있습니다. 중첩된 스코프는 기본적으로는 참조만 가능할 뿐, 대입은 되지 않는다는 것에 주의해야 합니다. 지역 변수들은 가장 내부의 스코프에서 읽고 씁니다. 마찬가지로, 전역 변수들은 전역 이름 공간에서 읽고 씁니다. `nonlocal`은 바깥 스코프에 쓰는 것을 허락합니다.

new-style class (뉴스타일 클래스)

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

object (객체)

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

optimized scope (최적화된 스코프)

A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package (패키지)

서브 모듈들이나, 재귀적으로 서브 패키지들을 포함할 수 있는 파이썬 모듈. 기술적으로, 패키지는 `__path__` 어트리뷰트가 있는 파이썬 모듈입니다.

정규 패키지 와 이름 공간 패키지 도 보세요.

parameter (매개변수)

함수 (또는 메서드) 정의에서 함수가 받을 수 있는 **인자** (또는 어떤 경우 인자들) 를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있습니다:

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

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

- 위치-전용 (*positional-only*): 위치로만 제공될 수 있는 인자를 지정합니다. 위치 전용 매개변수는 함수 정의의 매개변수 목록에 / 문자를 포함하고 그 뒤에 정의할 수 있습니다, 예를 들어 다음에서 *posonly1* 과 *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- 키워드-전용 (*keyword-only*): 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 *를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 *kw_only1* 와 *kw_only2*:

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

- 가변-위치 (*var-positional*): (다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정합니다. 이런 매개변수는 매개변수 이름에 * 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 다음에서 *args*:

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

- 가변-키워드 (*var-keyword*): (다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정합니다. 이런 매개변수는 매개변수 이름에 **를 앞에 붙여서 정의될 수 있습니다, 예를 들어 위의 예에서 *kwargs*.

매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있습니다.

인자 용어집 항목, 인자와 매개변수의 차이에 나오는 FAQ 질문, `inspect.Parameter` 클래스, `function` 절, [PEP 362](#)도 보세요.

path entry (경로 엔트리)

경로 기반 파인더 가 임포트 할 모듈들을 찾기 위해 참고하는 **임포트 경로** 상의 하나의 장소.

path entry finder (경로 엔트리 파인더)

`sys.path_hooks` 에 있는 콜러블 (즉, **경로 엔트리** 혹) 이 돌려주는 **파인더** 인데, 주어진 **경로 엔트리** 로 모듈을 찾는 방법을 알고 있습니다.

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

path entry hook (경로 엔트리 혹)

`sys.path_hooks` 리스트에 있는 콜러블인데, 특정 **경로 엔트리** 에서 모듈을 찾는 법을 알고 있다면 **경로 엔트리 파인더** 를 돌려줍니다.

path based finder (경로 기반 파인더)

기본 **메타 경로 파인더**들 중 하나인데, **임포트 경로** 에서 모듈을 찾습니다.

path-like object (경로류 객체)

파일 시스템 경로를 나타내는 객체. 경로류 객체는 경로를 나타내는 `str` 나 `bytes` 객체이거나 `os.PathLike` 프로토콜을 구현하는 객체입니다. `os.PathLike` 프로토콜을 지원하는 객체는 `os.fspath()` 함수를 호출해서 `str` 나 `bytes` 파일 시스템 경로로 변환될 수 있습니다; 대신 `os.fsdecode()` 와 `os.fsencode()` 는 각각 `str` 나 `bytes` 결과를 보장하는데 사용될 수 있습니다. [PEP 519](#)로 도입되었습니다.

PEP

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

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

PEP 1 참조하세요.

portion (포션)

PEP 420에서 정의한 것처럼, 이름 공간 패키지에 이바지하는 하나의 디렉토리에 들어있는 파일들의 집합 (zip 파일에 저장되는 것도 가능합니다).

positional argument (위치 인자)

인자를 보세요.

provisional API (잠정 API)

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

잠정 API에서조차도, 과거 호환성이 유지되지 않는 변경은 “최후의 수단”으로 여겨집니다 - 모든 식별된 문제들에 대해 과거 호환성을 유지하는 해법을 찾으려는 모든 시도가 선행됩니다.

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

provisional package (잠정 패키지)

잠정 API를 보세요.

Python 3000 (파이썬 3000)

파이썬 3.x 배포 라인의 별명 (버전 3의 배포가 먼 미래의 이야기던 시절에 만들어진 이름이다.) 이것을 “Py3k”로 줄여 쓰기도 합니다.

Pythonic (파이썬다운)

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

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

더 깔끔한, 파이썬다운 방법은 이렇습니다:

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

qualified name (정규화된 이름)

모듈의 전역 스코프에서 모듈에 정의된 클래스, 함수, 메서드에 이르는 “경로”를 보여주는 점으로 구분된 이름. **PEP 3155**에서 정의됩니다. 최상위 함수와 클래스의 경우에, 정규화된 이름은 객체의 이름과 같습니다:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

모듈을 가리키는데 사용될 때, 완전히 정규화된 이름 (*fully qualified name*)은 모든 부모 패키지들을 포함해서 모듈로 가는 점으로 분리된 이름을 의미합니다, 예를 들어, `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count (참조 횟수)

객체에 대한 참조의 개수. 객체의 참조 횟수가 0으로 떨어지면, 메모리가 반납됩니다. 일부 객체는 불멸이며 참조 횟수가 수정되지 않아서, 객체가 할당 해제되지 않습니다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소입니다. 프로그래머는 특정 객체의 참조 횟수를 돌려주는 `sys.getrefcount()` 함수를 호출할 수 있습니다.

regular package (정규 패키지)

`__init__.py` 파일을 포함하는 디렉터리와 같은 전통적인 패키지.

이름 공간 패키지 도 보세요.

REPL

An acronym for the “read-eval-print loop”, another name for the *interactive* interpreter shell.

__slots__

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

sequence (시퀀스)

`__getitem__()` 특수 메서드를 통해 정수 인덱스를 사용한 빠른 요소 액세스를 지원하고, 시퀀스의 길이를 돌려주는 `__len__()` 메서드를 정의하는 *이터러블*. 몇몇 내장 시퀀스들을 나열해보면, `list`, `str`, `tuple`, `bytes` 가 있습니다. `dict` 또한 `__getitem__()` 과 `__len__()` 을 지원하지만, 조회에 정수 대신 임의의 *해시 가능* 키를 사용하기 때문에 시퀀스가 아니라 매핑으로 취급된다는 것에 주의해야 합니다.

`collections.abc.Sequence` 추상 베이스 클래스는 `__getitem__()` 과 `__len__()` 을 넘어서 훨씬 풍부한 인터페이스를 정의하는데, `count()`, `index()`, `__contains__()`, `__reversed__()` 를 추가합니다. 이 확장된 인터페이스를 구현한 형을 `register()` 를 사용해서 명시적으로 등록할 수 있습니다. 시퀀스 메서드 일반에 대한 자세한 문서는, 공통 시퀀스 연산을 참조하세요.

set comprehension (집합 컴프리헨션)

이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 집합을 반환하는 간결한 방법. `results = {c for c in 'abracadabra' if c not in 'abc'}`는 문자열의 집합 `{'r', 'd'}`를 생성합니다. comprehensions을 참조하십시오.

single dispatch (싱글 디스패치)

구현이 하나의 인자의 형에 기초해서 결정되는 *제네릭 함수* 디스패치의 한 형태.

slice (슬라이스)

보통 *시퀀스*의 일부를 포함하는 객체. 슬라이스는 서브 스크립트 표기법을 사용해서 만듭니다. `variable_name[1:3:5]` 처럼, `[]` 안에서 여러 개의 숫자를 콜론으로 분리합니다. 대괄호 (서브 스크립트) 표기법은 내부적으로 `slice` 객체를 사용합니다.

soft deprecated (약하게 폐지된)

A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See [PEP 387: Soft Deprecation](#).

special method (특수 메서드)

파이썬이 형에 어떤 연산을, 덧셈 같은, 실행할 때 묵시적으로 호출되는 메서드. 이런 메서드는 두 개의 밑줄로 시작하고 끝나는 이름을 갖고 있습니다. 특수 메서드는 `specialnames`에 문서로 만들어져 있습니다.

statement (문장)

문장은 스위트 (코드의 “블록(block)”) 를 구성하는 부분입니다. 문장은 표현식 이거나 키워드를 사용하는 여러 가지 구조물 중의 하나입니다. 가령 `if`, `while`, `for`.

static type checker (정적 형 검사기)

An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also [type hints](#) and the `typing` module.

strong reference (강한 참조)

In Python’s C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

[빌린 참조](#) 도 보세요.

text encoding (텍스트 인코딩)

A string in Python is a sequence of Unicode code points (in range U+0000–U+10FFFF). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as “encoding”, and recreating the string from the sequence of bytes is known as “decoding”.

There are a variety of different text serialization codecs, which are collectively referred to as “text encodings”.

text file (텍스트 파일)

`str` 객체를 읽고 쓸 수 있는 [파일 객체](#). 종종, 텍스트 파일은 실제로는 바이트 지향 데이터스트림을 액세스하고 텍스트 인코딩을 자동 처리합니다. 텍스트 파일의 예로는 텍스트 모드 ('r' 또는 'w') 로 열린 파일, `sys.stdin`, `sys.stdout`, `io.StringIO` 의 인스턴스를 들 수 있습니다.

[바이트열류 객체](#) 를 읽고 쓸 수 있는 파일 객체에 대해서는 [바이너리 파일](#) 도 참조하세요.

thread state

The information used by the [CPython](#) runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be [attached](#) at once.

An [attached thread state](#) is required to call most of Python’s C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be [attached](#) in that thread at any given moment.

See Thread State and the Global Interpreter Lock for more information.

token (토큰)

A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python’s lexical analyzer. The `token` module contains information on the various types of tokens.

triple-quoted string (삼중 따옴표 된 문자열)

따옴표 (") 나 작은따옴표 (') 세 개로 둘러싸인 문자열. 그냥 따옴표 하나로 둘러싸인 문자열에 없는 기능을 제공하지는 않지만, 여러 가지 이유에서 쓸모가 있습니다. 이스케이프 되지 않은 작은따옴표나 큰따옴표를 문자열 안에 포함할 수 있도록 하고, 연결 문자를 쓰지 않고도 여러 줄에 걸쳐 쓸 수 있는데, 독스트링을 쓸 때 특히 쓸모 있습니다.

type (형)

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

type alias (형 에일리어스)

형을 식별자에 대입하여 만들어지는 형의 동의어.

형 에일리어스는 형 힌트를 단순화하는 데 유용합니다. 예를 들면:

```
def remove_gray_shades(
    colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
Color = tuple[int, int, int]

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

이 기능을 설명하는 typing과 **PEP 484**를 참조하세요.

type hint (형 힌트)

변수, 클래스 어트리뷰트 및 함수 매개변수 나 반환 값의 기대되는 형을 지정하는 어노테이션.

형 힌트는 선택 사항이고 파이썬에서 강제되지는 않지만, 정적 형 검사기에 유용합니다. 또한 IDE의 코드 완성 및 리팩토링을 돕습니다.

지역 변수를 제외하고, 전역 변수, 클래스 어트리뷰트 및 함수의 형 힌트는 typing.get_type_hints()를 사용하여 액세스할 수 있습니다.

이 기능을 설명하는 typing과 **PEP 484**를 참조하세요.

universal newlines (유니버설 줄 넘김)

다음과 같은 것들을 모두 줄의 끝으로 인식하는, 텍스트 스트림을 해석하는 태도: 유닉스 개행 문자 관례 '\n', 윈도우즈 관례 '\r\n', 예전의 매킨토시 관례 '\r'. 추가적인 사용에 관해서는 bytes.splitlines() 뿐만 아니라 **PEP 278**와 **PEP 3116**도 보세요.

variable annotation (변수 어노테이션)

변수 또는 클래스 어트리뷰트의 어노테이션.

변수 또는 클래스 어트리뷰트에 어노테이션을 달 때 대입은 선택 사항입니다:

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

변수 어노테이션은 일반적으로 형 힌트로 사용됩니다: 예를 들어, 이 변수는 int 값을 가질 것으로 기대됩니다:

```
count: int = 0
```

변수 어노테이션 문법은 섹션 **annassign**에서 설명합니다.

이 기능을 설명하는 함수 어노테이션, **PEP 484** 및 **PEP 526**을 참조하세요. 또한 어노테이션 작업에 대한 모범 사례는 annotations-howto를 참조하세요.

virtual environment (가상 환경)

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

venv도 보세요.

virtual machine (가상 기계)

소프트웨어만으로 정의된 컴퓨터. 파이썬의 가상 기계는 바이트 코드 컴파일러가 출력하는 바이트 코드를 실행합니다.

Zen of Python (파이썬 젠)

파이썬 디자인 원리와 철학들의 목록인데, 언어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 "import this"를 입력하면 보입니다.

이 설명서에 관하여

파이썬 설명서는 `reStructuredText` 소스에서 만들어진 것으로, 원래 파이썬을 위해 제작되었고 이제는 독립 프로젝트로 유지 관리되는 설명서 생성기인 `Sphinx` 를 사용했습니다.

설명서와 이를 위한 툴체인 개발은 파이썬 자체와 마찬가지로 전적으로 자원봉사자의 노력입니다. 기여하고 싶다면, 참여 방법에 대한 정보는 `reporting-bugs` 페이지를 참고하십시오. 새로운 자원봉사자는 언제나 환영합니다!

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 저자;
- `reStructuredText`와 `Docutils` 스위트를 만드는 `Docutils` 프로젝트.
- Fredrik Lundh, 그의 대안 파이썬 참조(Alternative Python Reference) 프로젝트에서 `Sphinx`가 많은 아이디어를 얻었습니다.

B.1 파이썬 설명서의 공헌자들

많은 사람이 파이썬 언어, 파이썬 표준 라이브러리 및 파이썬 설명서에 기여했습니다. 기여자의 부분적인 목록은 파이썬 소스 배포판의 `Misc/ACKS` 를 참조하십시오.

파이썬이 이런 멋진 설명서를 갖게 된 것은 파이썬 커뮤니티의 입력과 기여 때문입니다 - 감사합니다!

역사와 라이선스

C.1 소프트웨어의 역사

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

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

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

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

배포판	파생된 곳	해	소유자	GPL-compatible? (1)
0.9.0 ~ 1.2	n/a	1991-1995	CWI	yes
1.3 ~ 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 이상	2.1.1	2001-현재	PSF	yes

참고

- (1) GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-

compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

- (2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is "not incompatible" with the GPL.

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

파이썬에 포함된 일부 소프트웨어에는 다른 라이선스가 적용됩니다. 라이선스는 해당 라이선스에 해당하는 코드와 함께 나열됩니다. 이러한 라이선스의 불완전한 목록은 포함된 소프트웨어에 대한 라이선스 및 승인을 참조하십시오.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using this software ("Python") in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights Reserved" are retained in Python alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python.
4. PSF is making Python available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License

(다음 페이지에 계속)

(이전 페이지에서 계속)

Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

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

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and

(다음 페이지에 계속)

(이전 페이지에서 계속)

otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: <http://hdl.handle.net/1895.22/1013>".
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

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

C.3 포함된 소프트웨어에 대한 라이선스 및 승인

이 섹션은 파이썬 배포판에 포함된 제삼자 소프트웨어에 대한 불완전하지만 늘어나고 있는 라이선스와 승인의 목록입니다.

C.3.1 메르센 트위스터

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

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

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

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

Redistribution and use in source and binary forms, with or without
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

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

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 소켓

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

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

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

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

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL

(다음 페이지에 계속)

(이전 페이지에서 계속)

DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.3 비동기 소켓 서비스

The `test.support.asyncchat` and `test.support.asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

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

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

C.3.4 쿠키 관리

`http.cookies` 모듈은 다음과 같은 주의 사항을 포함합니다:

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

All Rights Reserved

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.5 실행 추적

trace 모듈은 다음과 같은 주의 사항을 포함합니다:

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

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

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

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

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

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

C.3.6 UUencode 및 UUdecode 함수

The uu codec contains the following notice:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

Modified by Jack Jansen, CWI, July 1995:

- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

C.3.7 XML 원격 프로시저 호출

xmlrpc.client 모듈은 다음과 같은 주의 사항을 포함합니다:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB

Copyright (c) 1999-2002 by Fredrik Lundh

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

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

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

C.3.8 test_epoll

The test.test_epoll module contains the following notice:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.9 Select kqueue

select 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

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

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

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

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

C.3.10 SipHash24

파일 Python/pyhash.c 에는 Dan Bernstein의 SipHash24 알고리즘의 Marek Majkowski의 구현이 포함되어 있습니다. 여기에는 다음과 같은 내용이 포함되어 있습니다:

```
<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

C.3.11 strtod 와 dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```

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

```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```

                        Apache License
                        Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction,
and distribution as defined by Sections 1 through 9 of this document.

"Licenser" shall mean the copyright owner or entity authorized by
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual,

(다음 페이지에 계속)

(이전 페이지에서 계속)

worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,

(다음 페이지에 계속)

(이전 페이지에서 계속)

any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

C.3.13 expat

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

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

Permission is hereby granted, free of charge, to any person obtaining

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

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

C.3.14 libffi

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

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

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

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

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

C.3.15 zlib

zlib 확장은 시스템에서 발견된 zlib 버전이 너무 오래되어서 빌드에 사용될 수 없으면, 포함된 zlib 소스 사본을 사용하여 빌드됩니다:

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

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

Jean-loup Gailly
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

C.3.16 cfuhash

tracemalloc 에 의해 사용되는 해시 테이블의 구현은 cfuhash 프로젝트를 기반으로 합니다:

Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

C.3.17 libmpdec

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

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.
```

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

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

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

C.3.18 W3C C14N 테스트 스위트

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at <https://www.w3.org/TR/xml-c14n2-testcases/> and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),  
All Rights Reserved.
```

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

- * Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,

(다음 페이지에 계속)

(이전 페이지에서 계속)

DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.19 mimalloc

MIT License:

Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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

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

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

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from `uvloop 0.16`, which is distributed under the MIT license:

Copyright (c) 2015-2021 MagicStack Inc. <http://magic.io>

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

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

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

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in `subr_smr.c`. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>
```

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

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

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


APPENDIX D

저작권

파이썬과 이 설명서는:

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

전체 라이선스 및 사용 권한 정보는 [역사와 라이선스](#) 에서 제공합니다.

알파벳 이외

..., 75

>>>, 75

__future__, 81

__slots__, 89

A

abstract base class (추상 베이스 클래스), 75

annotate function, 75

annotation (어노테이션), 75

argument

difference from parameter, 12

argument (인자), 75

asynchronous context manager (비동기 컨텍스트 관리자), 76

asynchronous generator (비동기 제너레이터), 76

asynchronous generator iterator (비동기 제너레이터 이터레이터), 76

asynchronous iterable (비동기 이터러블), 76

asynchronous iterator (비동기 이터레이터), 76

attached thread state, 76

attribute (어트리뷰트), 77

awaitable (어웨이터블), 77

B

BDFL, 77

binary file (바이너리 파일), 77

borrowed reference (빌린 참조), 77

bytecode (바이트 코드), 77

bytes-like object (바이트열류 객체), 77

C

callable (콜러블), 77

callback (콜백), 78

C-contiguous, 78

class (클래스), 78

class variable (클래스 변수), 78

closure variable (클로저 변수), 78

complex number (복소수), 78

context (컨텍스트), 78

context management protocol (컨텍스트 관리 프로토콜), 78

context manager (컨텍스트 관리자), 78

context variable (컨텍스트 변수), 78

contiguous (연속), 78

coroutine (코루틴), 79

coroutine function (코루틴 함수), 79

CPython, 79

current context (현재 컨텍스트), 79

D

decorator (데코레이터), 79

descriptor (디스크립터), 79

dictionary (딕셔너리), 79

dictionary comprehension (딕셔너리 컴프리헨션), 79

dictionary view (딕셔너리 뷰), 79

docstring (독스트링), 79

duck-typing (덕 타이핑), 80

E

EAFP, 80

evaluate function, 80

expression (표현식), 80

extension module (확장 모듈), 80

F

f-string (f-문자열), 80

file object (파일 객체), 80

file-like object (파일류 객체), 80

filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기), 80

finder (파인더), 80

floor division (정수 나눗셈), 81

Fortran contiguous, 78

free threading (자유 스레딩), 81

free variable (자유 변수), 81

function (함수), 81

function annotation (함수 어노테이션), 81

G

garbage collection (가비지 수거), 81

generator (제너레이터), 81

generator expression (제너레이터 표현식), 82

generator iterator (제너레이터 이터레이터), 81

generic function (제네릭 함수), [82](#)
generic type (제네릭 형), [82](#)
GIL, [82](#)
global interpreter lock (전역 인터프리터 록), [82](#)

H

hash-based pyc (해시 기반 *pyc*), [82](#)
hashable (해시 가능), [82](#)

I

IDLE, [82](#)
immortal (불멸), [83](#)
immutable (불변), [83](#)
import path (임포트 경로), [83](#)
importer (임포터), [83](#)
importing (임포트), [83](#)
interactive (대화형), [83](#)
interpreted (인터프리티드), [83](#)
interpreter shutdown (인터프리터 종료), [83](#)
iterable (이터러블), [83](#)
iterator (이터레이터), [83](#)

K

key function (키 함수), [84](#)
keyword argument (키워드 인자), [84](#)

L

lambda (람다), [84](#)
LBYL, [84](#)
lexical analyzer (어휘 분석기), [84](#)
list (리스트), [84](#)
list comprehension (리스트 컴프리헨션), [84](#)
loader (로더), [84](#)
locale encoding (로케일 인코딩), [85](#)

M

magic
 method (메서드), [85](#)
magic method (매직 메서드), [85](#)
mapping (매핑), [85](#)
meta path finder (메타 경로 파인더), [85](#)
metaclass (메타 클래스), [85](#)
method (메서드), [85](#)
 magic, [85](#)
 special, [89](#)
method resolution order (메서드 결정 순서), [85](#)
module (모듈), [85](#)
module spec (모듈 스펙), [85](#)
MRO, [85](#)
mutable (가변), [85](#)

N

named tuple (네임드 튜플), [85](#)
namespace (이름 공간), [86](#)
namespace package (이름 공간 패키지), [86](#)
nested scope (중첩된 스코프), [86](#)
new-style class (뉴스타일 클래스), [86](#)

O

object (객체), [86](#)
optimized scope (최적화된 스코프), [86](#)

P

package (패키지), [86](#)
parameter
 difference from argument, [12](#)
parameter (매개변수), [87](#)
PATH, [52](#)
path based finder (경로 기반 파인더), [87](#)
path entry (경로 엔트리), [87](#)
path entry finder (경로 엔트리 파인더), [87](#)
path entry hook (경로 엔트리 훅), [87](#)
path-like object (경로류 객체), [87](#)
PEP, [87](#)
portion (포션), [88](#)
positional argument (위치 인자), [88](#)
provisional API (잠정 *API*), [88](#)
provisional package (잠정 패키지), [88](#)
Python 3000 (파이썬 3000), [88](#)
Python 향상 제안
 PEP 1, [88](#)
 PEP 5, [5](#)
 PEP 8, [8](#), [33](#), [70](#)
 PEP 238, [81](#)
 PEP 278, [91](#)
 PEP 302, [84](#)
 PEP 343, [78](#)
 PEP 362, [76](#), [87](#)
 PEP 373, [4](#)
 PEP 387, [3](#)
 PEP 411, [88](#)
 PEP 420, [86](#), [88](#)
 PEP 443, [82](#)
 PEP 483, [82](#)
 PEP 484, [75](#), [81](#), [82](#), [91](#)
 PEP 492, [76](#), [77](#), [79](#)
 PEP 498, [80](#)
 PEP 519, [87](#)
 PEP 525, [76](#)
 PEP 526, [75](#), [91](#)
 PEP 572, [41](#)
 PEP 585, [82](#)
 PEP 602, [4](#)
 PEP 649, [75](#)
 PEP 683, [83](#)
 PEP 703, [56](#), [81](#), [82](#)
 PEP 3116, [91](#)
 PEP 3147, [35](#)
 PEP 3155, [88](#)

PYTHON_GIL, [82](#)
PYTHONDONTWRITEBYTECODE, [35](#)
Pythonic (파이썬다운), [88](#)

Q

qualified name (정규화된 이름), [88](#)

R

reference count (참조 횟수), [89](#)
regular package (정규 패키지), [89](#)
REPL, [89](#)

S

sequence (시퀀스), [89](#)
set comprehension (집합 컴프리헨션), [89](#)
single dispatch (싱글 디스패치), [89](#)
slice (슬라이스), [89](#)
soft deprecated (약하게 폐지된), [89](#)
special
 method (메서드), [89](#)
special method (특수 메서드), [89](#)
statement (문장), [90](#)
static type checker (정적 형 검사기), [90](#)
strong reference (강한 참조), [90](#)

T

text encoding (텍스트 인코딩), [90](#)
text file (텍스트 파일), [90](#)
thread state, [90](#)
token (토큰), [90](#)
triple-quoted string (삼중 따옴표 된 문자열),
 [90](#)
type (형), [90](#)
type alias (형 에일리어스), [91](#)
type hint (형 힌트), [91](#)

U

universal newlines (유니버설 줄 넘김), [91](#)

V

variable annotation (변수 어노테이션), [91](#)
virtual environment (가상 환경), [91](#)
virtual machine (가상 기계), [91](#)

Y

환경 변수
 PATH, [52](#)
 PYTHON_GIL, [82](#)
 PYTHONDONTWRITEBYTECODE, [35](#)

Z

Zen of Python (파이썬 젠), [91](#)