Community Experience Distilled

Expert Python Programming

Second Edition

Become an ace Python programmer by learning best coding practices and advance-level concepts with Python 3.5

Michał Jaworski
Tarek Ziadé

[PACKT] PUBLISHING open source community experience distilled

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 **'Current Status of Python'**
- A synopsis of the book's content
- More information on **Expert Python Programming Second Edition**

About the Authors

Michał Jaworski has 7 years of experience in Python. He is also the creator of graceful, which is a REST framework built on top of falcon. He has been in various roles at different companies: from an ordinary full-stack developer through software architect to VP of engineering in a fast-paced start-up company. He is currently a lead backend engineer in TV Store team at Opera Software. He is highly experienced in designing high-performance distributed services. He is also an active contributor to some of the popular Python open source projects.

Tarek Ziadé is an engineering manager at Mozilla, working with a team specialized in building web services in Python at scale for Firefox. He's contributed to the Python packaging effort and has worked with a lot of different Python web frameworks since Zope in the early days.

Tarek has also created Afpy, the French Python User Group, and has written two books on Python in French. He has delivered numerous talks and tutorials in French at international events such as Solutions Linux, PyCon, OSCON, and EuroPython.

Preface

Python rocks!

From the earliest version in the late 1980s to the current version, it has evolved with the same philosophy: providing a multiparadigm programming language with readability and productivity in mind.

People used to see Python as yet another scripting language and wouldn't feel right about using it to build large systems. However, over the years and thanks to some pioneer companies, it became obvious that Python could be used to build almost any kind of system.

In fact, many developers that come from another language are charmed by Python and make it their language of choice.

This is something you are probably aware of if you have bought this book, so there's no need to convince you about the merits of the language any further.

This book is written to express many years of experience of building all kinds of applications with Python, from small system scripts done in a couple of hours to very large applications written by dozens of developers over several years.

It describes the best practices used by developers when working with Python.

This book covers some topics that do not focus on the language itself but rather on the tools and techniques used to work with it.

In other words, this book describes how an advanced Python developer works every day.

What this book covers

Chapter 1, *Current Status of Python*, showcases the current state of the Python language and its community. It shows how Python is constantly changing, why it is changing, and also why these facts are important for anyone who wants to call themselves a Python professional. This chapter also features the most popular and canonical ways of working in Python—popular productivity tools and conventions that are de facto standards now.

Chapter 2, *Syntax Best Practices – below the Class Level*, presents iterators, generators, descriptors, and so on, in an advanced way. It also covers useful notes about Python idioms and internal CPython types implementations with their computational complexities as a rationale for showcased idioms.

Chapter 3, *Syntax Best Practices – above the Class Level*, explains syntax best practices, but focuses above the class level. It covers more advanced object-oriented concepts and mechanisms available in Python. This knowledge is required in order to understand the last section of the chapter, which presents different approaches to metaprogramming in Python.

Chapter 4, *Choosing Good Names*, involves choosing good names. It is an extension to PEP 8 with naming best practices, but also gives tips on designing good APIs.

Chapter 5, *Writing a Package*, explains how to create the Python package and which tools to use in order to properly distribute it on the official Python Package Index or any other package repository. Information about packages is supplemented with a brief review of the tools that allow you to create standalone executables from Python sources.

Chapter 6, *Deploying Code*, aims mostly at Python web developers and backend engineers, because it deals with code deployments. It explains how Python applications should be built in order to be easily deployed to remote servers and what tools you can use in order to automate that process. This chapter dovetails with *Chapter 5*, *Writing a Package*, because it shows how packages and private package repositories can be used to streamline your application deployments.

Chapter 7, *Python Extensions in Other Languages*, explains why writing C extensions for Python might be a good solution sometimes. It also shows that it is not as hard as it seems to be as long as the proper tools are used.

Chapter 8, *Managing Code*, gives some insight into how a project code base can be managed and explains how to set up various continuous development processes.

Chapter 9, *Documenting Your Project*, covers documentation and provides tips on technical writing and how Python projects should be documented.

Chapter 10, *Test-Driven Development*, explains the basic principles of test-driven development and the tools that can be used in this development methodology.

Chapter 11, *Optimization – General Principles and Profiling Techniques*, explains optimization. It provides profiling techniques and an optimization strategy guideline.

Chapter 12, *Optimization – Some Powerful Techniques*, extends *Chapter 11*, *Optimization – General Principles and Profiling Techniques*, by providing some common solutions to the performance problems that are often found in Python programs.

Chapter 13, *Concurrency*, introduces the vast topic of concurrency in Python. It explains what concurrency is, when it might be necessary to write concurrent applications, and what are the main approaches to concurrency for Python programmers.

Chapter 14, *Useful Design Patterns*, concludes the book with a set of useful design patterns and example implementations in Python.

1
Current Status of Python

Python is good for developers.

No matter what operating system you or your customers are running, it will work. Unless you are coding platform-specific things, or using a platform-specific library, you can work on Linux and deploy on other systems, for example. However, that's not uncommon anymore (Ruby, Java, and many other languages work in the same way). Combined with the other qualities that we will discover throughout this book, Python becomes a smart choice for a company's primary development language.

This book is focused on the latest version of Python, 3.5, and all code examples are written in this version of the language unless another version is explicitly mentioned. Because this release is not yet widely used, this chapter contains some description of the current *status quo* of Python 3 to introduce readers to it, as well as some introductory information on modern approaches to development in Python. This chapter covers the following topics:

- How to maintain compatibility between Python 2 and Python 3
- How to approach the problem of environment isolation both on application and operating system level for the purpose of development
- How to enhance the Python prompt
- How to install packages using pip

A book always starts with some appetizers. So, if you are already familiar with Python (especially with the latest 3.x branch) and know how to properly isolate environments for development purposes, you can skip the first two sections of this chapter and just read the other sections quickly. They describe some tools and resources that are not essential but can highly improve productivity in Python. Be sure to read the section on application-level environment isolation and pip, though, as their installation is mandatory for the rest of the book.

Where are we now and where we are going?

Python history starts somewhere in the late 1980s, but its 1.0 release date was in the year 1994, so it is not a very young language. There could be a whole timeline of major Python releases mentioned here, but what really matters is a single date: December 3, 2008 – the release date of Python 3.0.

At the time of writing, seven years have passed since the first Python 3 release. It is also four years since the creation of PEP 404—the official document that "un-released" Python 2.8 and officially closed the 2.x branch. Although a lot of time has passed, there is a specific dichotomy in the Python community—while the language develops very fast, there is a large group of its users that do not want to move forward with it.

Why and how does Python change?

The answer is simple—Python changes because there is such a need. The competition does not sleep. Every few months a new language pops out out of nowhere claiming to solve problems of all its predecessors. Most projects like these lose developers' attention very quickly and their popularity is driven by a sudden hype.

Anyway, this is a sign of some bigger issue. People design new languages because they find the existing ones unsuitable for solving their problems in the best ways possible. It would be silly not to recognize such a need. Also, more and more wide spread usage of Python shows that it could, and should, be improved in many places.

Lots of improvements in Python are often driven by the needs of particular fields where it is used. The most significant one is web development, which necessitated improvements to deal with concurrency in Python.

Some changes are just caused by the age and maturity of the Python project. Throughout the years, it has collected some of the clutter in the form of de-organized and redundant standard library modules or some bad design decisions. First, the Python 3 release aimed to bring major clean-up and refreshment to the language, but time showed that this plan backfired a bit. For a long time, it was treated by many developers only like curiosity, but, hopefully, this is changing.

Getting up to date with changes – PEP documents

The Python community has a well-established way of dealing with changes. While speculative Python language ideas are mostly discussed on specific mailing lists (`python-ideas@python.org`), nothing major ever gets changed without the existence of a new document called a PEP. A **PEP** is a **Python Enhancement Proposal**. It is a paper written that proposes a change on Python, and is a starting point for the community to discuss it. The whole purpose, format, and workflow around these documents is also standardized in the form of a Python Enhancement Proposal—precisely, PEP 1 document (`http://www.python.org/dev/peps/pep-0001`).

PEP documents are very important for Python and depending on the topic, they serve different purposes:

- **Informing**: They summarize the information needed by core Python developers and notify about Python release schedules
- **Standardizing**: They provide code style, documentation, or other guidelines
- **Designing**: They describe the proposed features

A list of all the proposed PEPs is available as in a document—PEP 0 (`https://www.python.org/dev/peps/`). Since they are easily accessible in one place and the actual URL is also very easy to guess, they are usually referred to by the number in the book.

Those who are wondering what the direction is in which the Python language is heading but do not have time to track a discussion on Python mailing lists, the PEP 0 document can be a great source of information. It shows which documents have already been accepted but are not yet implemented and also which are still under consideration.

PEPs also serve additional purposes. Very often, people ask questions like:

- Why does feature A work that way?
- Why does Python not have feature B?

In most such cases, the extensive answer is available in specific PEP documents where such a feature has already been mentioned. There are a lot of PEP documents describing Python language features that were proposed but not accepted. These documents are left as a historical reference.

Python 3 adoption at the time of writing this book

So, is Python 3, thanks to new exciting features, well adopted among its community? Sadly, not yet. The popular page Python 3 Wall of Superpowers (`https://python3wos.appspot.com`) that tracks the compatibility of most popular packages with the Python 3 branch was, until not so long ago, named Python 3 Wall of Shame. This situation is changing and the table of listed packages on the mentioned page is slowly turning "more green" with every month. Still, this does not mean that all teams building their applications will shortly use only Python 3. When all popular packages are available on Python 3, the popular excuse—the packages that we use are not ported yet—will no longer be valid.

The main reason for such a situation is that porting the existing application from Python 2 to Python 3 is always a challenge. There are tools like 2to3 that can perform automated code translation but they do not ensure that the result will be 100% correct. Also, such translated code may not perform as well as in its original form without manual adjustments. The moving of existing complex code bases to Python 3 might involve tremendous effort and cost that some organizations may not be able to afford. Still such costs can be split in time. Some good software architecture design methodologies, such as service-oriented architecture or microservices, can help to achieve this goal gradually. New project components (services or microservices) can be written using the new technology and existing ones can be ported one at a time.

In the long run, moving to Python 3 can only have beneficial effects on a project. According to PEP-404, there won't be a 2.8 release in the 2.x branch of Python anymore. Also, there may be a time in the future when all major projects such as Django, Flask, and numpy will drop any 2.x compatibility and will only be available on Python 3.

My personal opinion on this topic can be considered controversial. I think that the best incentive for the community would be to completely drop Python 2 support when creating new packages. This, of course, greatly limits the reach of such software but it may be the only way to change the way of thinking of those who insist on sticking to Python 2.x.

The main differences between Python 3 and Python 2

It has already been said that Python 3 breaks backwards compatibility with Python 2. Still, it is not a complete redesign. Also, it does not mean that every Python module written for a 2.x release will stop working under Python 3. It is possible to write completely cross-compatible code that will run on both major releases without additional tools or techniques, but usually it is possible only for simple applications.

Why should I care?

Despite my personal opinion on Python 2 compatibility, exposed earlier in this chapter, it is impossible to simply forget about it right at this time. There are still some useful packages (such as fabric, mentioned in *Chapter 6*, *Deploying the Code*) that are really worth using but are not likely to be ported in the very near future.

Also, sometimes we may be constrained by the organization we work in. The existing legacy code may be so complex that porting it is not economically feasible. So, even if we decide to move on and live only in the Python 3 world from now on, it will be impossible to completely live without Python 2 for some time.

Nowadays, it is very hard to name oneself a professional developer without giving something back to the community, so helping the open source developers in adding Python 3 compatibility to the existing packages is a good way to pay off the "moral debt" incurred by using them. This, of course, cannot be done without knowing the differences between Python 2 and Python 3. By the way, this is also a great exercise for those new in Python 3.

The main syntax differences and common pitfalls

The Python documentation is the best reference for differences between every release. Anyway, for readers' convenience, this section summarizes the most important ones. This does not change the fact that the documentation is mandatory reading for those not familiar with Python 3 yet (see `https://docs.python.org/3.0/whatsnew/3.0.html`).

The breaking changes introduced by Python 3 can generally be divided into a few groups:

- Syntax changes, wherein some syntax elements were removed/changed and other elements were added
- Changes in the standard library
- Changes in datatypes and collections

Syntax changes

Syntax changes that make it difficult for the existing code to run are the easiest to spot—they will cause the code to not run at all. The Python 3 code that features new syntax elements will fail to run on Python 2 and vice versa. The elements that are removed will make Python 2 code visibly incompatible with Python 3. The running code that has such issues will immediately cause the interpreter to fail raising a `SyntaxError` exception. Here is an example of the broken script that has exactly two statements, of which none will be executed due to the syntax error:

```
print("hello world")
print "goodbye python2"
```

Its actual result when run on Python 3 is as follows:

```
$ python3 script.py
  File "script.py", line 2
    print "goodbye python2"
                          ^
SyntaxError: Missing parentheses in call to 'print'
```

The list of such differences is a bit long and, from time to time, any new Python 3.x release may add new elements of syntax that will raise such errors on earlier releases of Python (even on the same 3.x branch). The most important of them are covered in *Chapter 2*, *Syntax Best Practices – below the Class Level*, and *Chapter 3*, *Syntax Best Practices – above the Class Level*, so there is no need to list all of them here.

The list of things dropped or changed from Python 2.7 is shorter, so here are the most important ones:

- `print` is no longer a statement but a function instead, so the parenthesis is now obligatory.
- Catching exceptions changed from `except exc, var` to `except exc as var`.
- The `<>` comparison operator has been removed in favor of `!=`.

- `from module import *` (https://docs.python.org/3.0/reference/simple_stmts.html#import) is now allowed only on a module level, no longer inside the functions.
- `from .[module] import name` is now the only accepted syntax for relative imports. All imports not starting with the dot character are interpreted as absolute imports.
- The `sort()` function and the list's `sorted()` method no longer accept the `cmp` argument. The `key` argument should be used instead.
- Division expressions on integers such as `1/2` return floats. The truncating behavior is achieved through the `//` operator like `1//2`. The good thing is that this can be used with floats too, so `5.0//2.0 == 2.0`.

Changes in the standard library

Breaking changes in the standard library are the second easiest to catch after syntax changes. Each subsequent version of Python adds, deprecates, improves, or completely removes standard library modules. Such a process was regular also in the older versions of Python (1.x and 2.x), so it does not come as a shock in Python 3.
In most cases, depending on the module that was removed or reorganized (like `urlparse` being moved to `urllib.parse`), it will raise exceptions on the import time just after it was interpreted. This makes such issues so easy to catch. Anyway, in order to be sure that all such issues are covered, the full test code coverage is essential.
In some cases (for example, when using lazily loaded modules), the issues that are usually noticed on import time will not appear before some modules are used in code as function calls. This is why, it is so important to make sure that every line of code is actually executed during tests suite.

Lazily loaded modules

A lazily loaded module is a module that is not loaded on import time. In Python, `import` statements can be included inside of functions so import will happen on a function call and not on import time. In some cases, such loading of modules may be a reasonable choice but in most cases, it is a workaround for poorly designed module structures (for example, to avoid circular imports) and should be generally avoided. For sure, there is no justifiable reason to lazily load standard library modules.

Changes in datatypes and collections

Changes in how Python represents datatypes and collections require the most effort when the developer tries to maintain compatibility or simply port existing code to Python 3. While incompatible syntax or standard library changes are easily noticeable and the most easy to fix, changes in collections and types are either nonobvious or require a lot of repetitive work. A list of such changes is long and, again, official documentation is the best reference.

Still, this section must cover the change in how string literals are treated in Python 3 because it seems to be the most controversial and discussed change in Python 3, despite being a very good thing that now makes things more explicit.

All string literals are now Unicode and `bytes` literals require a `b` or `B` prefix. For Python 3.0 and 3.1 using `u` prefix (like `u"foo"`) was dropped and will raise a syntax error. Dropping that prefix was the main reason for all controversies. It made really hard to create code that was compatible in different branches of Python—version 2.x relied on this prefix in order to create Unicode literals. This prefix was brought back in Python 3.3 to ease the integration process, although without any syntactic meaning.

The popular tools and techniques used for maintaining cross-version compatibility

Maintaining compatibility between versions of Python is a challenge. It may add a lot of additional work depending on the size of the project but is definitely doable and worth doing. For packages that are meant to be reused in many environments, it is an absolute must have. Open source packages without well-defined and tested compatibility bounds are very unlikely to become popular, but also, closed third-party code that never leaves the company network can greatly benefit from being tested in different environments.

It should be noted here that while this part focuses mainly on compatibility between various versions of Python, these approaches apply for maintaining compatibility with external dependencies like different package versions, binary libraries, systems, or external services.

The whole process can be divided into three main areas, ordered by importance:

- Defining and documenting target compatibility bounds and how they will be managed
- Testing in every environment and with every dependency version declared as compatible
- Implementing actual compatibility code

Declaration of what is considered compatible is the most important part of the whole process because it gives the users of the code (developers) the ability to have expectations and make assumptions on how it works and how it can change in the future. Our code can be used as a dependency in different projects that may also strive to manage compatibility, so the ability to reason how it behaves is crucial.

While this book tries to always give a few choices rather than to give an absolute recommendation on specific options, here is one of the few exceptions. The best way so far to define how compatibility may change in the future is by the proper approach to versioning numbers using *Semantic Versioning* (`http://semver.org/`), or shortly, semver. It describes a broadly accepted standard for marking the scope of change in code by the version specifier consisting only of three numbers. It also gives some advice on how to handle deprecation policies. Here is an excerpt from its summary:

Given a version number `MAJOR.MINOR.PATCH`, increment:

- A `MAJOR` version when you make incompatible API changes
- A `MINOR` version when you add functionality in a backwards-compatible manner
- A `PATCH` version when you make backwards-compatible bug fixes

Additional labels for pre-release and build metadata are available as extensions to the `MAJOR.MINOR.PATCH` format.

When it comes to testing, the sad truth is that to be sure that code is compatible with every declared dependency version and in every environment (here, the Python version), it must be tested in every combination of these. This, of course, may not be possible when the project has a lot of dependencies because the number of combinations grows rapidly with every new dependency in a version. So, typically some trade off needs to be made so that running full compatibility tests does not take ages. A selection of tools that help testing in so-called matrixes is presented in *Chapter 10*, *Test-Driven Development*, that discusses testing in general.

The benefit of using projects that follow semver is that usually what needs to be tested are only major releases because minor and patch releases are guaranteed not to include backwards incompatible changes. This is only true if such projects can be trusted not to break such a contract. Unfortunately, mistakes happen to everyone and backward incompatible changes happen in a lot of projects, even on patch versions. Still, since semver declares strict compatibility on minor and patch version changes, breaking it is considered a bug, so it may be fixed in patch release.

Implementation of the compatibility layer is last and also least important if bounds of that compatibility are well-defined and rigorously tested. Still there are some tools and techniques that every programmer interested in such a topic should know.

The most basic is Python's `__future__` module. It ports back some features from newer Python releases back into the older ones and takes the form of import statement:

```
from __future__ import <feature>
```

Features provided by `future` statements are syntax-related elements that cannot be easily handled by different means. This statement affects only the module where it was used. Here is an example of Python 2.7 interactive session that brings Unicode literals from Python 3.0:

```
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> type("foo")  # old literals
<type 'str'>
>>> from __future__ import unicode_literals
>>> type("foo")  # now is unicode
<type 'unicode'>
```

Here is a list of all the available `__future__` statement options that developers concerned with 2/3 compatibility should know:

- `division`: This adds a Python 3 division operator (PEP 238)
- `absolute_import`: This makes every form of `import` statement not starting with a dot character interpreted as an absolute import (PEP 328)
- `print_function`: This changes a `print` statement into a function call, so parentheses around `print` becomes mandatory (PEP 3112)
- `unicode_literals`: This makes every string literal interpreted as Unicode literals (PEP 3112)

A list of the `__future__` statement options is very short and it covers only a few syntax features. The other things that have changed like the metaclass syntax (which is an advanced feature covered in *Chapter 3*, *Syntax Best Practices – above the Class Level*), are a lot harder to maintain. Reliably handling of multiple standard library reorganizations also cannot be solved by `future` statements. Happily, there are some tools that aim to provide a consistent layer of ready-to-use compatibility. The most commonly known is Six (`https://pypi.python.org/pypi/six/`) that provides whole common 2/3 compatibility boilerplate as a single module. The other promising but slightly less popular tool is the future module (`http://python-future.org/`).

In some situations, developers may not want to include additional dependencies in some small packages. A common practice is the additional module that gathers all the compatibility code, usually named `compat.py`. Here is an example of such a `compat` module taken from the `python-gmaps` project (`https://github.com/swistakm/python-gmaps`):

```
# -*- coding: utf-8 -*-
import sys

if sys.version_info < (3, 0, 0):
    import urlparse  # noqa

    def is_string(s):
        return isinstance(s, basestring)

else:
    from urllib import parse as urlparse  # noqa

    def is_string(s):
        return isinstance(s, str)
```

Such a `compat.py` module is popular even in projects that depends on Six for 2/3 compatibility because it is a very convenient way to store code that handles compatibility with different versions of packages used as dependencies.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

- Log in or register to our website using your e-mail address and password.
- Hover the mouse pointer on the **SUPPORT** tab at the top.
- Click on **Code Downloads & Errata**.
- Enter the name of the book in the **Search** box.
- Select the book for which you're looking to download the code files.
- Choose from the drop-down menu where you purchased this book from.
- Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Expert-Python-Programming_Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Not only CPython

The main Python implementation is written in the C language and is called **CPython**. It is the one that the majority of people refer to when they talk about Python. When the language evolves, the C implementation is changed accordingly. Besides C, Python is available in a few other implementations that are trying to keep up with the mainstream. Most of them are a few milestones behind CPython, but provide a great opportunity to use and promote the language in a specific environment.

Why should I care?

There are plenty of alternative Python implementations available. The Python Wiki page on that topic (`https://wiki.python.org/moin/PythonImplementations`) features more than 20 different language variants, dialects, or implementations of Python interpreter built with something else than C. Some of them implement only a subset of the core language syntax, features, and built-in extensions but there is at least a few that are almost fully compatible with CPython. The most important thing to know is that while some of them are just toy projects or experiments, most of them were created to solve some real problems – problems that were either impossible to solve with CPython or required too much of the developer's effort. Examples of such problems are:

- Running Python code on embedded systems
- Integration with code written for runtime frameworks such as Java or .NET or in different languages
- Running Python code in web browsers

This section provides a short description of subjectively most popular and up-to-date choices that are currently available for Python programmers.

Stackless Python

Stackless Python advertises itself as an enhanced version of Python. Stackless is named so because it avoids depending on the C call stack for its own stack. It is in fact a modified CPython code that also adds some new features that were missing from core Python implementation at the time Stackless was created. The most important of them are microthreads managed by the interpreter as a cheap and lightweight alternative to ordinary threads that must depend on system kernel context switching and tasks scheduling.

The latest available versions are 2.7.9 and 3.3.5 that implement 2.7 and 3.3 versions of Python respectively. All the additional features provided by Stackless are exposed as a framework within this distribution through the built-in `stackless` module.

Stackless isn't the most popular alternative implementation of Python, but it is worth knowing because ideas introduced in it have had a strong impact on the language community. The core switching functionality was extracted from Stackless and published as an independent package named `greenlet`, which is now a basis for many useful libraries and frameworks. Also, most of its features were re-implemented in PyPy—another Python implementation that will be featured later. Refer to `http://stackless.readthedocs.org/`.

Jython

Jython is a Java implementation of the language. It compiles the code into Java byte code, and allows the developers to seamlessly use Java classes within their Python modules. Jython allows people to use Python as the top-level scripting language on complex application systems, for example, J2EE. It also brings Java applications into the Python world. Making Apache Jackrabbit (which is a document repository API based on JCR; see `http://jackrabbit.apache.org`) available in a Python program is a good example of what Jython allows.

The latest available version of Jython is Jython 2.7, and this corresponds to 2.7 version of the language. It is advertised as implementing nearly all of the core Python standard library and uses the same regression test suite. The version of Jython 3.x is under development.

The main differences of Jython as compared to CPython implementation are:

- True Java's garbage collection instead of reference counting
- The lack of **GIL** (**global interpreter lock**) allows a better utilization of multiple cores in multi-threaded applications

The main weakness of this implementation of the language is the lack of support for C Python Extension APIs, so no Python extensions written in C will work with Jython. This might change in the future because there are plans to support the C Python Extension API in Jython 3.x.

Some Python web frameworks such as Pylons were known to be boosting Jython development to make it available in the Java world. Refer to `http://www.jython.org`.

IronPython

IronPython brings Python into the .NET Framework. The project is supported by Microsoft, where IronPython's lead developers work. It is quite an important implementation for the promotion of a language. Besides Java, the .NET community is one of the biggest developer communities out there. It is also worth noting that Microsoft provides a set of free development tools that turn Visual Studio into full-fledged Python IDE. This is distributed as Visual Studio plugins named **PVTS** (**Python Tools for Visual Studio**) and is available as open source code on GitHub (`http://microsoft.github.io/PTVS`).

The latest stable release is 2.7.5 and it is compatible with Python 2.7. Similar to Jython, there is some development around Python 3.x implementation, but there is no stable release available yet. Despite the fact that .NET runs primarily on Microsoft Windows, it is possible to run IronPython also on Mac OS X and Linux. This is thanks to Mono, a cross platform, open source .NET implementation.

Main differences or advantages of IronPython as compared to CPython are as follows:

- Similar to Jython, the lack of GIL (global interpreter lock) allows the better utilization of multiple cores in multi-threaded applications
- Code written in C# and other .NET languages can be easily integrated in IronPython and vice versa
- Can be run in all major web browsers through Silverlight

When speaking about weaknesses, IronPython, again, seems very similar to Jython because it does not support the C Python Extension APIs. This is important for developers who would like to use packages such as numpy that are largely based on C extensions. There is a project called ironclad (refer to `https://github.com/IronLanguages/ironclad`) that aims to allow using such extensions seamlessly with IronPython, albeit its last known supported release is 2.6 and development seems to have stopped at this point. Refer to `http://ironpython.net/`.

PyPy

PyPy is probably the most exciting implementation, as its goal is to rewrite Python into Python. In PyPy, the Python interpreter is itself written in Python. We have a C code layer carrying out the nuts-and-bolts work for the CPython implementation of Python. However, in the PyPy implementation, this C code layer is rewritten in pure Python.

This means you can change the interpreter's behavior during execution time and implement code patterns that couldn't be easily done in CPython.

PyPy currently aims to be fully compatible with Python 2.7, while PyPy3 is compatible with Python 3.2.5 version.

In the past, PyPy was interesting mostly for theoretical reasons, and it interested those who enjoyed going deep into the details of the language. It was not generally used in production, but this has changed through the years. Nowadays, many benchmarks show that surprisingly PyPy is often way faster than the CPython implementation. This project has its own benchmarking site that tracks the performance of each version measured using tens of different benchmarks (refer to `http://speed.pypy.org/`). It clearly shows that PyPy with JIT enabled is at least a few times faster than CPython. This and other features of PyPy makes more and more developers decide to switch to PyPy in their production environments.

The main differences of PyPy as compared to the CPython implementation are:

- Garbage collection is used instead of reference counting
- Integrated tracing JIT compiler that allows impressive improvements in performance
- Application-level Stackless features are borrowed from Stackless Python

Like almost every other alternative Python implementation, PyPy lacks the full official support of C Python Extension API. Still it, at least, provides some sort of support for C extensions through its CPyExt subsystem, although it is poorly documented and still not feature complete. Also, there is an ongoing effort within the community in porting NumPy to PyPy because it is the most requested feature. Refer to `http://pypy.org`.

Modern approaches to Python development

A deep understanding of the programming language of choice is the most important thing to harness as an expert. This will always be true for any technology. Still, it is really hard to develop a good software without knowing the common tools and practices within the given language community. Python has no single feature that could not be found in some other language. So, in direct comparison of syntax, expressiveness, or performance, there will always be a solution that is better in one or more fields. But the area in which Python really stands out from the crowd is in the whole ecosystem built around the language. Its community has, for years, polished the standard practices and libraries that help to create more reliable software in a shorter time.

The most obvious and important part of the mentioned ecosystem is a huge collection of free and open source packages that solve a multitude of problems. Writing new software is always an expensive and time-consuming process. Being able to reuse the existing code instead of *reinventing the wheel* greatly reduces the time and costs of development. For some companies, it is the only reason their projects are economically feasible.

Due to this reason, Python developers put a lot of effort on creating tools and standards to work with open source packages created by others. Starting from virtual isolated environments, improved interactive shells and debuggers, to programs that help to discover, search, and analyze the huge collection of packages available on **PyPI** (**Python Package Index**).

Application-level isolation of Python environments

Nowadays, a lot of operating systems come with Python as a standard component. Most Linux distributions and Unix-based systems such as FreeBSD, NetBSD, OpenBSD, or OS X come with Python are either installed by default or available through system package repositories. Many of them even use it as part of some core components—Python powers the installers of Ubuntu (Ubiquity), Red Hat Linux (Anaconda), and Fedora (Anaconda again).

Due to this fact, a lot of packages from PyPI are also available as native packages managed by the system's package management tools such as `apt-get` (Debian, Ubuntu), `rpm` (Red Hat Linux), or `emerge` (Gentoo). Although it should be remembered that the list of available libraries is very limited and they are mostly outdated when compared to PyPI. This is the reason why `pip` should always be used to obtain new packages in the latest version as a recommendation of **PyPA** (**Python Packaging Authority**). Although it is an independent package starting from version 2.7.9 and 3.4 of CPython, it is bundled with every new release by default. Installing the new package is as simple as this:

```
pip install <package-name>
```

Among other features, `pip` allows forcing specific versions of packages (using the `pip install package-name==version` syntax) and upgrading to the latest version available (using the `--upgrade` switch). The full usage description for most of the command-line tools presented in the book can be easily obtained simply by running the command with the `-h` or `--help` switch, but here is an example session that demonstrates the most commonly used options:

```
$ pip show pip
---
Metadata-Version: 2.0
Name: pip
Version: 7.1.2
Summary: The PyPA recommended tool for installing Python packages.
Home-page: https://pip.pypa.io/
Author: The pip developers
Author-email: python-virtualenv@groups.google.com
License: MIT
Location: /usr/lib/python2.7/site-packages
Requires:

$ pip install 'pip<7.0.0'
Collecting pip<7.0.0
  Downloading pip-6.1.1-py2.py3-none-any.whl (1.1MB)
    100% |████████████████████████████████| 1.1MB 242kB/s
Installing collected packages: pip
  Found existing installation: pip 7.1.2
    Uninstalling pip-7.1.2:
      Successfully uninstalled pip-7.1.2
Successfully installed pip-6.1.1
You are using pip version 6.1.1, however version 7.1.2 is available.
You should consider upgrading via the 'pip install --upgrade pip'
command.

$ pip install --upgrade pip
You are using pip version 6.1.1, however version 7.1.2 is available.
You should consider upgrading via the 'pip install --upgrade pip'
command.
```

```
Collecting pip
  Using cached pip-7.1.2-py2.py3-none-any.whl
Installing collected packages: pip
  Found existing installation: pip 6.1.1
    Uninstalling pip-6.1.1:
      Successfully uninstalled pip-6.1.1
Successfully installed pip-7.1.2
```

In some cases, `pip` may not be available by default. From Python 3.4 version onwards (and also Python 2.7.9), it can always be bootstrapped using the `ensurepip` module:

```
$ python -m ensurepip
Ignoring indexes: https://pypi.python.org/simple
Requirement already satisfied (use --upgrade to upgrade): setuptools in /
usr/lib/python2.7/site-packages
Collecting pip
Installing collected packages: pip
Successfully installed pip-6.1.1
```

The most up-to-date information on how to install pip for older Python versions is available on the project's documentation page at `https://pip.pypa.io/en/stable/installing/`.

Why isolation?

`pip` may be used to install system-wide packages. On Unix-based and Linux systems, this will require super user privileges, so the actual invocation will be:

```
sudo pip install <package-name>
```

Note that this is not required on Windows since it does not provide the Python interpreter by default, and Python on Windows is usually installed manually by the user without super user privileges.

Anyway, installing system-wide packages directly from PyPI is not recommended and should be avoided. This may seem like a contradiction with the previous statement that using `pip` is a PyPA recommendation, but there are some serious reasons for that. As explained earlier, Python is very often an important part of many packages available through operating system package repositories and may power a lot of important services. System distribution maintainers put a lot of effort in selecting the correct versions of packages to match various package dependencies. Very often, Python packages that are available from system's package repositories contain custom patches or are kept outdated only to ensure compatibility with some other system components. Forcing an update of such a package using `pip` to a version that breaks some backwards compatibility might break some crucial system services.

Doing such things only on the local computer for development purposes is also not a good excuse. Recklessly using `pip` that way is almost always asking for trouble and will eventually lead to issues that are very hard to debug. This does not mean that installing packages from PyPI globally is a strictly forbidden thing, but it should always be done consciously and while knowing the related risks.

Fortunately, there is an easy solution to this problem—environment isolation. There are various tools that allow the isolation of the Python runtime environment at different levels of system abstraction. The main idea is to isolate project dependencies from packages required by different projects and/or system services. The benefits of this approach are:

- It solves the "Project X depends on version 1.x but Project Y needs 4.x" dilemma. The developer can work on multiple projects with different dependencies that may even collide without the risk of affecting each other.
- The project is no longer constrained by versions of packages that are provided in his system distribution repositories.
- There is no risk of breaking other system services that depend on certain package versions because new package versions are only available inside such an environment.
- A list of packages that are project dependencies can be easily "frozen", so it is very easy to reproduce them.

The easiest and most lightweight approach to isolation is to use application-level virtual environments. They focus only on isolating the Python interpreter and packages available in it. They are very easy to set up and are very often just enough to ensure proper isolation during the development of small projects and packages.

Unfortunately, in some cases, this may not be enough to ensure enough consistency and reproducibility. For such cases, system-level isolation is a good addition to the workflow and some available solutions to that are explained later in this chapter.

Popular solutions

There are several ways to isolate Python at runtime. The simplest and most obvious, although hardest to maintain, is to manually change `PATH` and `PYTHONPATH` environment variables and/or move Python binary to a different place to affect the way it discovers available packages and change it to a custom place where we want to store our project's dependencies. Fortunately, there are several tools available that help in maintaining virtual environments and how installed packages are stored in the system. These are mainly: `virtualenv`, `venv`, and `buildout`. What they do under the hood is in fact the same as what we would do manually. The actual strategy depends on the specific tool implementation, but generally, they are more convenient to use and can provide additional benefits.

virtualenv

Virtualenv is by far the most popular tool in this list. Its name simply stands for Virtual Environment. It's not a part of the standard Python distribution, so it needs to be obtained using `pip`. It is one of the packages that is worth installing system-wide (using `sudo` on Linux and Unix-based systems).

Once it is installed, a new virtual environment is created using the following command:

```
virtualenv ENV
```

Here, `ENV` should be replaced by the desired name for the new environment. This will create a new `ENV` directory in the current working directory path. It will contain a few new directories inside:

- `bin/`: This is where the new Python executable and scripts/executables provided by other packages are stored.
- `lib/` and `include/`: These directories contain the supporting library files for the new Python inside the virtual environment. The new packages will be installed in `ENV/lib/pythonX.Y/site-packages/`.

Once the new environment is created, it needs to be activated in the current shell session using Unix's source command:

```
source ENV/bin/activate
```

This changes the state of the current shell sessions by affecting its environment variables. In order to make the user aware that he has activated the virtual environment, it will change the shell prompt by appending the `(ENV)` string at its beginning. Here is an example session that creates a new environment and activates it to illustrate this:

```
$ virtualenv example
New python executable in example/bin/python
Installing setuptools, pip, wheel...done.
$ source example/bin/activate
(example)$ deactivate
$
```

The important thing to note about `virtualenv` is that it depends completely on its state stored on a filesystem. It does not provide any additional abilities to track what packages should be installed in it. These virtual environments are not portable and should not be moved to another system/machine. This means that the new virtual environment needs to be created from scratch for each new application deployment. Because of that, there is a good practice used by `virtualenv` users to store all project dependencies in the `requirements.txt` file (this is the naming convention), as shown in the following code:

```
# lines followed by hash (#) are treated as a comments

# strict version names are best for reproducibility
eventlet==0.17.4
graceful==0.1.1

# for projects that are well tested with different
# dependency versions the relative version specifiers
# are acceptable too
falcon>=0.3.0,<0.5.0

# packages without versions should be avoided unless
# latest release is always required/desired
pytz
```

With such files, all dependencies can be easily installed using `pip` because it accepts the requirements file as its output:

```
pip install -r requirements.txt
```

What needs to be remembered is that the requirements file is not always the ideal solution because it does not define the exact list of dependencies, only those that are to be installed. So, the whole project can work without problems in a development environment but will fail to start in others if the requirements file is outdated and does not reflect actual state of environment. There is, of course, the `pip freeze` command that prints all packages in the current environment but it should not be used blindly—it will output everything, even packages that are not used in the project but installed only for testing. The other tool mentioned in the book, `buildout`, addresses this issue, so it may be a better choice for some development teams.

For Windows users, `virtualenv` under Windows uses a different naming for its internal structure of directories. You need to use `Scripts/`, `Libs/`, and `Include/` instead of `bin/`, `lib/`, `include/`, to better match development conventions on that operating system. The commands used for activating/deactivating the environment are also different; you need to use `ENV/Scripts/activate.bat` and `ENV/Scripts/deactivate.bat` instead of using `source` on `activate` and `deactivate` scripts.

venv

Virtual environments shortly became well established and a popular tool within the community. Starting from Python 3.3, creating virtual environments is supported by standard library. The usage is almost the same as with Virtualenv, although command-line options have quite a different naming convention. The new `venv` module provides a `pyvenv` script for creating a new virtual environment:

```
pyvenv ENV
```

Here, `ENV` should be replaced by the desired name for the new environment. Also, new environments can now be created directly from Python code because all functionality is exposed from the built-in `venv` module. The other usage and implementation details, like the structure of the environment directory and activate/deactivate scripts are mostly the same as in Virtualenv, so migration to this solution should be easy and painless.

For developers using newer versions of Python, it is recommended to use `venv` instead of Virtualenv. For Python 3.3, switching to `venv` may require more effort because in this version, it does not install `setuptools` and `pip` by default in the new environment, so the users need to install them manually. Fortunately, it has changed in Python 3.4, and also due to the customizability of `venv`, it is possible to override its behavior. The details are explained in the Python documentation (refer to `https://docs.python.org/3.5/library/venv.html`), but some users might find it too tricky and will stay with Virtualenv for that specific version of Python.

buildout

Buildout is a powerful tool for bootstrapping and the deployment of applications written in Python. Some of its advanced features will also be explained later in the book. For a long time, it was also used as a tool to create isolated Python environments. Because Buildout requires a declarative configuration that must be changed every time there is a change in dependencies, instead of relying on the environment state, these environments were easier to reproduce and manage.

Unfortunately, this has changed. The `buildout` package since version 2.0.0 no longer tries to provide any level of isolation from system Python installation. Isolation handling is left to other tools such as Virtualenv, so it is still possible to have isolated Buildouts, but things become a bit more complicated. A Buildout must be initialized inside an isolated environment in order to be really isolated.

This has a major drawback as compared to the previous versions of Buildout, since it depends on other solutions for isolation. The developer working on this code can no longer be sure whether the dependencies description is complete because some packages can be installed by bypassing the declarative configuration. This issue can of course be solved using proper testing and release procedures, but it adds some more complexity to the whole workflow.

To summarize, Buildout is no longer a solution that provides environment isolation but its declarative configuration can improve maintainability and the reproducibility of virtual environments.

Which one to choose?

There is no best solution that will fit every use case. What is good in one organization may not fit the workflow of other teams. Also, every application has different needs. Small projects can easily depend on sole `virtualenv` or `venv` but bigger ones may require additional help of `buildout` to perform more complex assembly.

What was not described in detail earlier is that previous versions of Buildout (buildout<2.0.0) allowed the assembly of projects in an isolated environment with similar results as provided by Virtualenv. Unfortunately, 1.x branch of this project is no longer maintained, so using it for that purpose is discouraged.

I would recommend to use `venv` module instead of Virtualenv whenever it is possible. So, this should be the default choice for projects targeting Python versions 3.4 and higher. Using `venv` in Python 3.3 may be a little inconvenient due to a lack of built-in support for `setuptools` and `pip`. For projects targeting a wider spectrum of Python run times (including alternative interpreters and 2.x branch), it seems that Virtualenv is the best choice.

System-level environment isolation

In most cases, software implementation can iterate fast because developers reuse a lot of existing components. Don't Repeat Yourself—this is a popular rule and motto of many programmers. Using other packages and modules to include them in the codebase is only a part of that culture. What also can be considered under "reused components" are binary libraries, databases, system services, third-party APIs, and so on. Even whole operating systems should be considered as reused.

Backend services of web-based applications are a great example of how complex such applications can be. The simplest software stack usually consists of a few layers (starting from the lowest):

- A database or other kind of storage
- The application code implemented in Python
- An HTTP server such as Apache or NGINX

Of course such stack can be even simpler but it is very unlikely. In fact, big applications are often so complex that it is hard to distinguish single layers. Big applications can use many different databases, be divided into multiple independent processes, and use many other system services for caching, queuing, logging, service discovery, and so on. Sadly, there are no limits for complexity and it seems that code simply follows the second law of thermodynamics.

What really is important is that not all of the software stack elements can be isolated on the level of Python runtime environment. No matter whether it is an HTTP server such as NGINX or RDBMS such as PostgreSQL, they are usually available in different versions on different systems. Making sure that everyone in a development team uses the same versions of every component is very hard without proper tools. It is theoretically possible that all developers in a team working on a single project will be able to get the same versions of services on their development boxes. But all this effort is futile if they do not use the same operating system as in the production environment. And forcing a programmer to work on something else other than his beloved system of choice is impossible for sure.

The problem lies in the fact that portability is still a big challenge. Not all services will work in exactly the same way in production environments as they do on the developer's machines and that is very unlikely to change. Even Python can behave differently on different systems despite how much work is put in to make it cross-platform. Usually, this is well documented and happens only in places that depend directly on system calls, but relying on the programmer's ability to remember a long list of compatibility quirks is quite an error prone strategy.

A popular solution to this problem is by isolating whole systems as application environments. This is usually achieved by leveraging different types of system virtualization tools. Virtualization, of course, reduces performance, but with modern computers that have hardware support for virtualization, the performance loss is usually negligible. On the other hand, a list of possible gains is very long:

- The development environment can exactly match the system version and services used in production, which helps in solving compatibility issues
- Definitions for system configuration tools such as Puppet, Chef, or Ansible (if used) can be reused for configuration of the development environment
- The newly hired team members can easily hop into the project if the creation of such environments is automated
- The developers can work directly with low system-level features that may not be available on operating systems they use for work, for example, **FUSE** (**File System in User Space**) that is not available in Windows

Virtual development environments using Vagrant

Vagrant currently seems to be the most popular tool that provides a simple and convenient way to create and manage development environments. It is available for Windows, Mac OS, and a few popular Linux distributions (refer to `https://www.vagrantup.com`). It does not have any additional dependencies. Vagrant creates new development environments in the form of virtual machines or containers. The exact implementation depends on a choice of virtualization providers. VirtualBox is the default provider and it is bundled with the Vagrant installer but additional providers are available as well. The most notable choices are VMware, Docker, LXC (Linux Containers), and Hyper-V.

The most important configuration is provided to Vagrant in a single file named `Vagrantfile`. It should be independent for every project. The following are the most important things it provides:

- Choice of virtualization provider
- Box used as a virtual machine image
- Choice of provisioning method
- Shared storage between a VM and a VM's host
- Ports that need to be forwarded between a VM and its host

Syntax language for the `Vagrantfile` is Ruby. The example configuration file provides a good template to start the project and has an excellent documentation, so the knowledge of this language is not required. Template configuration can be created using a single command:

```
vagrant init
```

This will create a new file named `Vagrantfile` in the current working directory. The best place to store this file is usually the root of the related project sources. This file is already a valid configuration that will create a new VM using the default provider and base box image. No provisioning is enabled by default. After the addition of `Vagrantfile`, the new VM is started using:

```
vagrant up
```

The initial start can take a few minutes because the actual box must be downloaded from the Web. There is also some initialization process that may take some time depending on the used provider, box, and system performance every time the already existing VM is brought up. Usually, this takes only a couple of seconds. Once the new Vagrant environment is up and running, developers can connect to SSH using this shorthand:

```
vagrant ssh
```

This can be done anywhere in the project source tree below the location of `Vagrantfile`. For developers' convenience, we will look in the directories above for the configuration file and match it with the related VM instance. Then, it establishes the secure shell connection, so the development environment can be interacted with like any ordinary remote machine. The only difference is that the whole project source tree (root defined as a location of `Vagrantfile`) is available on the VM's filesystem under `/vagrant/`.

Containerization versus virtualization

Containers are an alternative to full machine virtualization. It is a lightweight method of virtualization, where the kernel and operating system allow the running of multiple isolated user space instances. OS is shared between containers and host, so it theoretically requires less overhead than in full virtualization. Such a container contains only application code and its system-level dependencies, but from the perspective of processes running inside, it looks like a completely isolated system environment.

Software containers got their popularity mostly thanks to Docker; that is one of the available implementations. Docker allows to describe its container in the form of a simple text document called `Dockerfile`. Containers from such definitions can be built and stored. It also supports incremental changes, so if new things are added to the container then it does not need to be recreated from scratch.

Different tools such as Docker and Vagrant seem to overlap in features but the main difference between them is the reason why these tools were built. Vagrant, as mentioned earlier, is built primarily as a tool for development. It allows to bootstrap the whole virtual machine with a single command, but does not allow to simply pack it and deploy or release as is. Docker, on the other hand, is built exactly for that—preparing complete containers that can be sent and deployed to production as a whole package. If implemented well, this can greatly improve the process of product deployment. Because of that, using Docker and similar solutions (Rocket, for example) during development makes sense only if it also has to be used in the deployment process on production. Using it only for isolation purposes during development may generate too much overhead and also has a drawback of not being consistent.

Popular productivity tools

A productivity tool is a bit of a vague term. On one hand, almost every open source code package released and available online is a kind of productivity booster—it provides ready-to-use solutions to some problem, so no one needs to spend time on it (ideally speaking). On the other hand, one could say that the whole of Python is about productivity. And both are undoubtedly true. Almost everything in this language and community surrounding it seems to be designed in order to make software development as productive as it is possible.

This creates a positive feedback loop. Since writing code is fun and easy, a lot of programmers spend their free time to create tools that make it even easier and fun. And this fact will be used here as a basis for a very subjective and non-scientific definition of a productivity tool—a piece of software that makes development easier and more fun.

By nature, productivity tools focus mainly on certain elements of the development process such as testing, debugging, and managing packages and are not core parts of products that they help to build. In some cases, they may not even be referred to anywhere in the project's codebase despite being used on a daily basis.

The most important productivity tools, `pip` and `venv`, were already discussed earlier in this chapter. Some of them have packages for specific problems, such as profiling and testing, and have their own chapters in the book. This section is dedicated to other tools that are really worth mentioning, but have no specific chapter in the book where they could be introduced.

Custom Python shells – IPython, bpython, ptpython, and so on

Python programmers spend a lot of time in interactive interpreter sessions. It is very good for testing small code snippets, accessing documentation, or even debugging code at run time. The default interactive Python session is very simple and does not provide many features such as tab completion or code introspection helpers. Fortunately, the default Python shell can be easily extended and customized.

The interactive prompt can be configured with a startup file. When it starts, it looks for the `PYTHONSTARTUP` environment variable and executes the code in the file pointed to by this variable. Some Linux distributions provide a default startup script, which is generally located in your home directory. It is called `.pythonstartup`. Tab completion and command history are often provided to enhance the prompt and are based on the `readline` module. (You need the `readline` library.)

If you don't have such a file, you can easily create one. Here's an example of the simplest startup file that adds completion with the `<Tab>` key and history:

```
# python startup file
import readline
import rlcompleter
import atexit
import os

# tab completion
readline.parse_and_bind('tab: complete')

# history file
histfile = os.path.join(os.environ['HOME'], '.pythonhistory')
try:
    readline.read_history_file(histfile)

except IOError:
    pass

atexit.register(readline.write_history_file, histfile)
del os, histfile, readline, rlcompleter
```

Create this file in your home directory and call it `.pythonstartup`. Then, add a `PYTHONSTARTUP` variable in your environment using the path of your file:

Setting up the PYTHONSTARTUP environment variable

If you are running Linux or Mac OS X, the simplest way is to create the startup script in your home folder. Then, link it with a `PYTHONSTARTUP` environment variable set into the system shell startup script. For example, the Bash and Korn shells use the `.profile` file, where you can insert a line as follows:

```
export PYTHONSTARTUP=~/.pythonstartup
```

If you are running Windows, it is easy to set a new environment variable as an administrator in the system preferences, and save the script in a common place instead of using a specific user location.

Writing on the `PYTHONSTARTUP` script may be a good exercise but creating good custom shell all alone is a challenge that only few can find time for. Fortunately, there are a few custom Python shell implementations that immensely improve the experience of interactive sessions in Python.

IPython

IPyhton (`http://ipython.scipy.org`) provides an extended Python command shell. Among the features provided, the most interesting ones are:

- Dynamic object introspection
- System shell access from the prompt
- Profiling direct support
- Debugging facilities

Now, IPython is a part of the larger project called Jupyter that provides interactive notebooks with live code that can be written in many different languages.

bpython

bpython (`http://bpython-interpreter.org/`) advertises itself as a fancy interface to the python interpreter. Here are some of the accented on the projects page:

- In-line syntax highlighting
- Readline-like autocomplete with suggestions displayed as you type
- Expected parameter lists for any Python function
- Autoindentation
- Python 3 support

ptpython

ptpython (`https://github.com/jonathanslenders/ptpython/`) is another approach to the topic of advanced Python shells. In this project, core prompt utilities implementation is available as a separate package called `prompt_toolkit` (from the same author). This allows you to easily create various aesthetically pleasing interactive command-line interfaces.

It is often compared to bpython in functionalities but the main difference is that it enables a compatibility mode with IPython and its syntax that enables additional features such as `%pdb`, `%cpaste`, or `%profile`.

Interactive debuggers

Code debugging is an integral element of the software development process. Many programmers can spend most of their life using only extensive logging and `print` statements as their primary debugging tools but most professional developers prefer to rely on some kind of debugger.

Python already ships with a built-in interactive debugger called `pdb` (refer to `https://docs.python.org/3/library/pdb.html`). It can be invoked from the command line on the existing script, so Python will enter post-mortem debugging if the program exits abnormally:

```
python -m pdb script.py
```

Post-mortem debugging, while useful, does not cover every scenario. It is useful only when the application exists with some exception if the bug occurs. In many cases, faulty code just behaves abnormally but does not exit unexpectedly. In such cases, custom breakpoints can be set on a specific line of code using this single-line idiom:

```
import pdb; pdb.set_trace()
```

This will cause the Python interpreter to start the debugger session on this line during run time.

`pdb` is very useful for tracing issues and at first glance, it may look very familiar to the well-known GDB (GNU Debugger). Because Python is a dynamic language, the `pdb` session is very similar to an ordinary interpreter session. This means that the developer is not limited to tracing code execution but can call any code and even perform module imports.

Sadly, because of its roots (`bdb`), the first experience with `pdb` can be a bit overwhelming due to the existence of cryptic short letter debugger commands such as `h`, `b`, `s`, `n`, `j`, and `r`. Whenever in doubt, the `help pdb` command typed during the debugger session will provide extensive usage and additional information.

The debugger session in pdb is also very simple and does not provide additional features like tab completion or code highlighting. Fortunately, there are few packages available on PyPI that provide such features available from alternative Python shells mentioned in the previous section. The most notable examples are:

- `ipdb`: This is a separate package based on `ipython`
- `ptpdb`: This is a separate package based on `ptpython`
- `bpdb`: This is bundled with `bpython`

Useful resources

The Web is full of useful resources for Python developers. The most important and obvious ones were already mentioned earlier but here they are repeated to keep this list consistent:

- Python documentation
- PyPI—Python Package Index
- PEP 0—Index of Python Enhancement Proposals

The other resources such as books and tutorials are useful but often get outdated very fast. What does not get outdated are the resources that are actively curated by the community or released periodically. The two that are mostly worth recommending are:

- Awesome-python (`https://github.com/vinta/awesome-python`), which includes a curated list of popular packages and frameworks
- Python Weekly (`http://www.pythonweekly.com/`) is a popular newsletter that delivers to its subscribers dozens of new and interesting Python packages and resources every week

These two resources will provide the reader with tons of additional reading for several months.

Summary

This chapter started with topic differences between Python 2 and 3 with advice on how to deal with the current situation where a big part of its community is torn between two worlds. Then, it came to the modern approaches to Python development that were surprisingly developed mostly due to this unfortunate split between two major versions of the language. These are mostly different solutions to the environment isolation problem. The chapter ended with a short summary of the popular productivity tools as well as popular resources for further reference.

Where to buy this book

You can buy Expert Python Programming Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.