Python Guide Documentation

Release 0.0.1

Kenneth Reitz

February 17, 2017

Contents

1 Getting Started with Python **3**
- 1.1 Picking an Interpreter . . . 3
- 1.2 Properly Installing Python . . . 5
- 1.3 Installing Python on Mac OS X . . . 5
- 1.4 Installing Python on Windows . . . 7
- 1.5 Installing Python on Linux . . . 8

2 Writing Great Python Code **11**
- 2.1 Structuring Your Project . . . 11
- 2.2 Code Style . . . 22
- 2.3 Reading Great Code . . . 31
- 2.4 Documentation . . . 32
- 2.5 Testing Your Code . . . 35
- 2.6 Logging . . . 39
- 2.7 Common Gotchas . . . 42
- 2.8 Choosing a License . . . 45

3 Scenario Guide for Python Applications **47**
- 3.1 Network Applications . . . 47
- 3.2 Web Applications & Frameworks . . . 48
- 3.3 HTML Scraping . . . 54
- 3.4 Command-line Applications . . . 56
- 3.5 GUI Applications . . . 57
- 3.6 Databases . . . 59
- 3.7 Networking . . . 60
- 3.8 Systems Administration . . . 61
- 3.9 Continuous Integration . . . 66
- 3.10 Speed . . . 67
- 3.11 Scientific Applications . . . 73
- 3.12 Image Manipulation . . . 75
- 3.13 Data Serialization . . . 77
- 3.14 XML parsing . . . 77
- 3.15 JSON . . . 78
- 3.16 Cryptography . . . 79
- 3.17 Machine Learning . . . 80
- 3.18 Interfacing with C/C++ Libraries . . . 82

4 Shipping Great Python Code **85**

4.1 Packaging Your Code . . . 85
4.2 Freezing Your Code . . . 87

5 Python Development Environments **91**
5.1 Your Development Environment . . . 91
5.2 Virtual Environments . . . 96
5.3 Further Configuration of Pip and Virtualenv . . . 99

6 Additional Notes **101**
6.1 Introduction . . . 101
6.2 The Community . . . 102
6.3 Learning Python . . . 103
6.4 Documentation . . . 108
6.5 News . . . 109
6.6 Contribute . . . 110
6.7 License . . . 111
6.8 The Guide Style Guide . . . 111

Greetings, Earthling! Welcome to The Hitchhiker's Guide to Python.

This is a living, breathing guide. If you'd like to contribute, fork us on GitHub!

This handcrafted guide exists to provide both novice and expert Python developers a best practice handbook to the installation, configuration, and usage of Python on a daily basis.

This guide is **opinionated** in a way that is almost, but not quite, entirely *unlike* Python's official documentation. You won't find a list of every Python web framework available here. Rather, you'll find a nice concise list of highly recommended options.

Let's get started! But first, let's make sure you know where your towel is.

CHAPTER 1

Getting Started with Python

New to Python? Let's properly setup up your Python environment.

Picking an Interpreter

The State of Python (3 & 2)

When choosing a Python interpreter, one looming question is always present: "Should I choose Python 2 or Python 3"? The answer is a bit more subtle than one might think.

The basic gist of the state of things is as follows:

1. Most production applications today use Python 2.7.
2. Python 3 is ready for the production deployment of applications today.
3. Python 2.7 will only receive necessary security updates until 2020 [6].
4. The brand name "Python" encapsulates both Python 3 and Python 2.

Recommendations

I'll be blunt:

- Use Python 3 for new Python applications.
- If you're learning Python for the first time, familiarizing yourself with Python 2.7 will be very useful, but not more useful than learning Python 3.
- Learn both. They are both "Python".
- Software that is already built often depends on Python 2.7.
- If you are writing a new open source Python library, it's best to write it for both Python 2 and 3 simultaneously. Only supporting Python 3 for a new library you want to be widely adopted is a political statement and will alienate many of your users. This is not a problem — slowly, over the next three years, this will become less the case.

[6] https://www.python.org/dev/peps/pep-0373/#id2

So.... 3?

If you're choosing a Python interpreter to use, I recommend you use the newest Python 3.x, since every version brings new and improved standard library modules, security and bug fixes.

Given such, only use Python 2 if you have a strong reason to, such as a pre-existing code-base, a Python 2 exclusive library, simplicity/familiarity, or, of course, you absolutely love and are inspired by Python 2. No harm in that.

Check out Can I Use Python 3? to see if any software you're depending on will block your adoption of Python 3.

Further Reading

It is possible to write code that works on Python 2.6, 2.7, and Python 3. This ranges from trivial to hard depending upon the kind of software you are writing; if you're a beginner there are far more important things to worry about.

Implementations

When people speak of *Python* they often mean not just the language but also the CPython implementation. *Python* is actually a specification for a language that can be implemented in many different ways.

CPython

CPython is the reference implementation of Python, written in C. It compiles Python code to intermediate bytecode which is then interpreted by a virtual machine. CPython provides the highest level of compatibility with Python packages and C extension modules.

If you are writing open-source Python code and want to reach the widest possible audience, targeting CPython is best. To use packages which rely on C extensions to function, CPython is your only implementation option.

All versions of the Python language are implemented in C because CPython is the reference implementation.

PyPy

PyPy is a Python interpreter implemented in a restricted statically-typed subset of the Python language called RPython. The interpreter features a just-in-time compiler and supports multiple back-ends (C, CLI, JVM).

PyPy aims for maximum compatibility with the reference CPython implementation while improving performance.

If you are looking to increase performance of your Python code, it's worth giving PyPy a try. On a suite of benchmarks, it's currently over 5 times faster than CPython.

PyPy supports Python 2.7. PyPy3 [1], released in beta, targets Python 3.

Jython

Jython is a Python implementation that compiles Python code to Java bytecode which is then executed by the JVM (Java Virtual Machine). Additionally, it is able to import and use any Java class like a Python module.

If you need to interface with an existing Java codebase or have other reasons to need to write Python code for the JVM, Jython is the best choice.

Jython currently supports up to Python 2.7. [2]

[1] http://pypy.org/compat.html
[2] https://hg.python.org/jython/file/412a8f9445f7/NEWS

IronPython

IronPython is an implementation of Python for the .NET framework. It can use both Python and .NET framework libraries, and can also expose Python code to other languages in the .NET framework.

Python Tools for Visual Studio integrates IronPython directly into the Visual Studio development environment, making it an ideal choice for Windows developers.

IronPython supports Python 2.7. [3]

PythonNet

Python for .NET is a package which provides near seamless integration of a natively installed Python installation with the .NET Common Language Runtime (CLR). This is the inverse approach to that taken by IronPython (see above), to which it is more complementary than competing with.

In conjunction with Mono, pythonnet enables native Python installations on non-Windows operating systems, such as OS X and Linux, to operate within the .NET framework. It can be run in addition to IronPython without conflict.

Pythonnet supports from Python 2.6 up to Python 3.5. [4] [5]

- Properly Install Python

1.2 Properly Installing Python

There's a good chance that you already have Python on your operating system.

If so, you do not need to install or configure anything else to use Python. Having said that, I would strongly recommend that you install the tools and libraries described in the guides below before you start building Python applications for real-world use. In particular, you should always install Setuptools, Pip, and Virtualenv — they make it much easier for you to use other third-party Python libraries.

Installation Guides

These guides go over the proper installation of *Python* for development purposes, as well as setuptools, pip and virtualenv.

- Python 3 on MacOS.
- *Python 2 on MacOS*.
- *Python 2 on Microsoft Windows*.
- *Python 2 on Ubuntu Linux*.

Installing Python on Mac OS X

Note: Check out our guide for installing Python 3 on OS X.

[3] http://ironpython.codeplex.com/releases/view/81726
[4] https://travis-ci.org/pythonnet/pythonnet
[5] https://ci.appveyor.com/project/TonyRoberts/pythonnet-480xs

The latest version of Mac OS X, Sierra, **comes with Python 2.7 out of the box**.

You do not need to install or configure anything else to use Python. Having said that, I would strongly recommend that you install the tools and libraries described in the next section before you start building Python applications for real-world use. In particular, you should always install Setuptools, as it makes it much easier for you to install and manage other third-party Python libraries.

The version of Python that ships with OS X is great for learning but it's not good for development. The version shipped with OS X may be out of date from the official current Python release, which is considered the stable production version.

Doing it Right

Let's install a real version of Python.

Before installing Python, you'll need to install a C compiler. The fastest way is to install the Xcode Command Line Tools by running `xcode-select --install`. You can also download the full version of Xcode from the Mac App Store, or the minimal but unofficial OSX-GCC-Installer package.

Note: If you already have XCode installed, do not install OSX-GCC-Installer. In combination, the software can cause issues that are difficult to diagnose.

Note: If you perform a fresh install of XCode, you will also need to add the commandline tools by running `xcode-select --install` on the terminal.

While OS X comes with a large number of UNIX utilities, those familiar with Linux systems will notice one key component missing: a decent package manager. Homebrew fills this void.

To install Homebrew, open `Terminal` or your favorite OSX terminal emulator and run

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/insta
```

The script will explain what changes it will make and prompt you before the installation begins. Once you've installed Homebrew, insert the Homebrew directory at the top of your `PATH` environment variable. You can do this by adding the following line at the bottom of your `~/.profile` file

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Now, we can install Python 2.7:

```
$ brew install python
```

or Python 3:

```
$ brew install python3
```

This will take a minute or two.

Setuptools & Pip

Homebrew installs Setuptools and `pip` for you.

Setuptools enables you to download and install any compliant Python software over a network (usually the Internet) with a single command (`easy_install`). It also enables you to add this network installation capability to your own Python software with very little work.

`pip` is a tool for easily installing and managing Python packages, that is recommended over `easy_install`. It is superior to `easy_install` in several ways, and is actively maintained.

Virtual Environments

A Virtual Environment (commonly referred to as a 'virtualenv') is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the "Project X depends on version 1.x but, Project Y needs 4.x" dilemma, and keeps your global site-packages directory clean and manageable.

For example, you can work on a project which requires Django 1.10 while also maintaining a project which requires Django 1.8.

To start using this and see more information: *Virtual Environments* docs.

This page is a remixed version of another guide, which is available under the same license.

Installing Python on Windows

First, download the latest version of Python 2.7 from the official Website. If you want to be sure you are installing a fully up-to-date version, click the Downloads > Windows link from the home page of the Python.org web site .

The Windows version is provided as an MSI package. To install it manually, just double-click the file. The MSI package format allows Windows administrators to automate installation with their standard tools.

By design, Python installs to a directory with the version number embedded, e.g. Python version 2.7 will install at `C:\Python27\`, so that you can have multiple versions of Python on the same system without conflicts. Of course, only one interpreter can be the default application for Python file types. It also does not automatically modify the `PATH` environment variable, so that you always have control over which copy of Python is run.

Typing the full path name for a Python interpreter each time quickly gets tedious, so add the directories for your default Python version to the `PATH`. Assuming that your Python installation is in `C:\Python27\`, add this to your `PATH`:

```
C:\Python27\;C:\Python27\Scripts\
```

You can do this easily by running the following in `powershell`:

```
[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\;C:\Python27\Scripts\", "Us
```

The second (`Scripts`) directory receives command files when certain packages are installed, so it is a very useful addition. You do not need to install or configure anything else to use Python. Having said that, I would strongly recommend that you install the tools and libraries described in the next section before you start building Python applications for real-world use. In particular, you should always install Setuptools, as it makes it much easier for you to use other third-party Python libraries.

Setuptools + Pip

The most crucial third-party Python software of all is Setuptools, which extends the packaging and installation facilities provided by the distutils in the standard library. Once you add Setuptools to your Python system you can download and install any compliant Python software product with a single command. It

also enables you to add this network installation capability to your own Python software with very little work.

To obtain the latest version of Setuptools for Windows, run the Python script available here: ez_setup.py

You'll now have a new command available to you: **easy_install**. It is considered by many to be deprecated, so we will install its replacement: **pip**. Pip allows for uninstallation of packages, and is actively maintained, unlike easy_install.

To install pip, run the Python script available here: get-pip.py

Virtual Environments

A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the "Project X depends on version 1.x but, Project Y needs 4.x" dilemma, and keeps your global site-packages directory clean and manageable.

For example, you can work on a project which requires Django 1.10 while also maintaining a project which requires Django 1.8.

To start using this and see more information: *Virtual Environments* docs.

This page is a remixed version of another guide, which is available under the same license.

Installing Python on Linux

The latest versions of CentOS, Fedora, Redhat Enterprise (RHEL) and Ubuntu **come with Python 2.7 out of the box**.

To see which version of Python you have installed, open a command prompt and run

```
$ python --version
```

Some older versions of RHEL and CentOS come with Python 2.4 which is unacceptable for modern Python development. Fortunately, there are Extra Packages for Enterprise Linux which include high quality additional packages based on their Fedora counterparts. This repository contains a Python 2.6 package specifically designed to install side-by-side with the system's Python 2.4 installation.

You do not need to install or configure anything else to use Python. Having said that, I would strongly recommend that you install the tools and libraries described in the next section before you start building Python applications for real-world use. In particular, you should always install Setuptools and pip, as it makes it much easier for you to use other third-party Python libraries.

Setuptools & Pip

The two most crucial third-party Python packages are setuptools and pip.

Once installed, you can download, install and uninstall any compliant Python software product with a single command. It also enables you to add this network installation capability to your own Python software with very little work.

Python 2.7.9 and later (on the python2 series), and Python 3.4 and later include pip by default.

To see if pip is installed, open a command prompt and run

```
$ command -v pip
```

To install pip, follow the official pip installation guide - this will automatically install the latest version of setuptools.

Virtual Environments

A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the "Project X depends on version 1.x but, Project Y needs 4.x" dilemma, and keeps your global site-packages directory clean and manageable.

For example, you can work on a project which requires Django 1.10 while also maintaining a project which requires Django 1.8.

To start using this and see more information: *Virtual Environments* docs.

You can also use *virtualenvwrapper* to make it easier to manage your virtual environments.

CHAPTER 2

Writing Great Python Code

This part of the guide focuses on the best-practices for writing Python code.

Structuring Your Project

By "structure" we mean the decisions you make concerning how your project best meets its objective. We need to consider how to best leverage Python's features to create clean, effective code. In practical terms, "structure" means making clean code whose logic and dependencies are clear as well as how the files and folders are organized in the filesystem.

Which functions should go into which modules? How does data flow through the project? What features and functions can be grouped together and isolated? By answering questions like these you can begin to plan, in a broad sense, what your finished product will look like.

In this section we take a closer look at Python's module and import systems as they are the central elements to enforcing structure in your project. We then discuss various perspectives on how to build code which can be extended and tested reliably.

Structure of the Repository

It's Important.

Just as Code Style, API Design, and Automation are essential for a healthy development cycle, Repository structure is a crucial part of your project's architecture.

When a potential user or contributor lands on your repository's page, they see a few things:

- Project Name
- Project Description
- Bunch O' Files

Only when they scroll below the fold will the user see your project's README.

If your repo is a massive dump of files or a nested mess of directories, they might look elsewhere before even reading your beautiful documentation.

> Dress for the job you want, not the job you have.

Of course, first impressions aren't everything. You and your colleagues will spend countless hours working with this repository, eventually becoming intimately familiar with every nook and cranny. The layout of it is important.

Sample Repository

tl;dr: This is what Kenneth Reitz recommends.

This repository is available on GitHub.

```
README.rst
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py
```

Let's get into some specifics.

The Actual Module

Location	`./sample/` or `./sample.py`
Purpose	The code of interest

Your module package is the core focus of the repository. It should not be tucked away:

```
./sample/
```

If your module consists of only a single file, you can place it directly in the root of your repository:

```
./sample.py
```

Your library does not belong in an ambiguous src or python subdirectory.

License

Location	`./LICENSE`
Purpose	Lawyering up.

This is arguably the most important part of your repository, aside from the source code itself. The full license text and copyright claims should exist in this file.

If you aren't sure which license you should use for your project, check out choosealicense.com.

Of course, you are also free to publish code without a license, but this would prevent many people from potentially using your code.

Setup.py

Location	`./setup.py`
Purpose	Package and distribution management.

If your module package is at the root of your repository, this should obviously be at the root as well.

Requirements File

Location	`./requirements.txt`
Purpose	Development dependencies.

A pip requirements file should be placed at the root of the repository. It should specify the dependencies required to contribute to the project: testing, building, and generating documentation.

If your project has no development dependencies, or you prefer development environment setup via `setup.py`, this file may be unnecessary.

Documentation

Location	`./docs/`
Purpose	Package reference documentation.

There is little reason for this to exist elsewhere.

Test Suite

Location	`./test_sample.py` or `./tests`
Purpose	Package integration and unit tests.

Starting out, a small test suite will often exist in a single file:

```
./test_sample.py
```

Once a test suite grows, you can move your tests to a directory, like so:

```
tests/test_basic.py
tests/test_advanced.py
```

Obviously, these test modules must import your packaged module to test it. You can do this a few ways:

- Expect the package to be installed in site-packages.
- Use a simple (but *explicit*) path modification to resolve the package properly.

I highly recommend the latter. Requiring a developer to run `setup.py develop` to test an actively changing codebase also requires them to have an isolated environment setup for each instance of the codebase.

To give the individual tests import context, create a tests/context.py file:

```
import os
import sys
sys.path.insert(0, os.path.abspath('..'))

import sample
```

Then, within the individual test modules, import the module like so:

```
from .context import sample
```

This will always work as expected, regardless of installation method.

Some people will assert that you should distribute your tests within your module itself – I disagree. It often increases complexity for your users; many test suites often require additional dependencies and runtime contexts.

Makefile

Location	`./Makefile`
Purpose	Generic management tasks.

If you look at most of my projects or any Pocoo project, you'll notice a Makefile laying around. Why? These projects aren't written in C... In short, make is a incredibly useful tool for defining generic tasks for your project.

Sample Makefile:

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

test:
    py.test tests

.PHONY: init test
```

Other generic management scripts (e.g. `manage.py` or `fabfile.py`) belong at the root of the repository as well.

Regarding Django Applications

I've noticed a new trend in Django applications since the release of Django 1.4. Many developers are structuring their repositories poorly due to the new bundled application templates.

How? Well, they go to their bare and fresh repository and run the following, as they always have:

```
$ django-admin.py startproject samplesite
```

The resulting repository structure looks like this:

```
README.rst
samplesite/manage.py
samplesite/samplesite/settings.py
samplesite/samplesite/wsgi.py
samplesite/samplesite/sampleapp/models.py
```

Don't do this.

Repetitive paths are confusing for both your tools and your developers. Unnecessary nesting doesn't help anybody (unless they're nostalgic for monolithic SVN repos).

Let's do it properly:

```
$ django-admin.py startproject samplesite .
```

Note the "`.`".

The resulting structure:

```
README.rst
manage.py
samplesite/settings.py
samplesite/wsgi.py
samplesite/sampleapp/models.py
```

Structure of Code is Key

Thanks to the way imports and modules are handled in Python, it is relatively easy to structure a Python project. Easy, here, means that you do not have many constraints and that the module importing model is easy to grasp. Therefore, you are left with the pure architectural task of crafting the different parts of your project and their interactions.

Easy structuring of a project means it is also easy to do it poorly. Some signs of a poorly structured project include:

- Multiple and messy circular dependencies: if your classes Table and Chair in `furn.py` need to import Carpenter from `workers.py` to answer a question such as `table.isdoneby()`, and if conversely the class Carpenter needs to import Table and Chair, to answer the question `carpenter.whatdo()`, then you have a circular dependency. In this case you will have to resort to fragile hacks such as using import statements inside methods or functions.
- Hidden coupling: each and every change in Table's implementation breaks 20 tests in unrelated test cases because it breaks Carpenter's code, which requires very careful surgery to adapt the change. This means you have too many assumptions about Table in Carpenter's code or the reverse.
- Heavy usage of global state or context: instead of explicitly passing `(height, width, type, wood)` to each other, Table and Carpenter rely on global variables that can be modified and are modified on the fly by different agents. You need to scrutinize all access to these global variables to understand why a rectangular table became a square, and discover that remote template code is also modifying this context, messing with table dimensions.
- Spaghetti code: multiple pages of nested if clauses and for loops with a lot of copy-pasted procedural code and no proper segmentation are known as spaghetti code. Python's meaningful indentation (one of its most controversial features) make it very hard to maintain this kind of code. So the good news is that you might not see too much of it.
- Ravioli code is more likely in Python: it consists of hundreds of similar little pieces of logic, often classes or objects, without proper structure. If you never can remember if you have to use FurnitureTable, AssetTable or Table, or even TableNew for your task at hand, you might be swimming in ravioli code.

Modules

Python modules are one of the main abstraction layers available and probably the most natural one. Abstraction layers allow separating code into parts holding related data and functionality.

For example, a layer of a project can handle interfacing with user actions, while another would handle low-level manipulation of data. The most natural way to separate these two layers is to regroup all interfacing functionality in one file, and all low-level operations in another file. In this case, the interface file needs to import the low-level file. This is done with the `import` and `from ... import` statements.

As soon as you use *import* statements you use modules. These can be either built-in modules such as *os* and *sys*, third-party modules you have installed in your environment, or your project's internal modules.

To keep in line with the style guide, keep module names short, lowercase, and be sure to avoid using special symbols like the dot (.) or question mark (?). So a file name like `my.spam.py` is one you should avoid! Naming this way will interfere with the way Python looks for modules.

In the case of *my.spam.py* Python expects to find a `spam.py` file in a folder named `my` which is not the case. There is an example of how the dot notation should be used in the Python docs.

If you'd like you could name your module `my_spam.py`, but even our friend the underscore should not be seen often in module names.

Aside from some naming restrictions, nothing special is required for a Python file to be a module, but you need to understand the import mechanism in order to use this concept properly and avoid some issues.

Concretely, the `import modu` statement will look for the proper file, which is `modu.py` in the same directory as the caller if it exists. If it is not found, the Python interpreter will search for `modu.py` in the "path" recursively and raise an ImportError exception if it is not found.

Once `modu.py` is found, the Python interpreter will execute the module in an isolated scope. Any top-level statement in `modu.py` will be executed, including other imports if any. Function and class definitions are stored in the module's dictionary.

Then, the module's variables, functions, and classes will be available to the caller through the module's namespace, a central concept in programming that is particularly helpful and powerful in Python.

In many languages, an `include file` directive is used by the preprocessor to take all code found in the file and 'copy' it into the caller's code. It is different in Python: the included code is isolated in a module namespace, which means that you generally don't have to worry that the included code could have unwanted effects, e.g. override an existing function with the same name.

It is possible to simulate the more standard behavior by using a special syntax of the import statement: `from modu import *`. This is generally considered bad practice. **Using `import *` makes code harder to read and makes dependencies less compartmentalized**.

Using `from modu import func` is a way to pinpoint the function you want to import and put it in the global namespace. While much less harmful than `import *` because it shows explicitly what is imported in the global namespace, its only advantage over a simpler `import modu` is that it will save a little typing.

Very bad

```
[...]
from modu import *
[...]
x = sqrt(4)  # Is sqrt part of modu? A builtin? Defined above?
```

Better

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt may be part of modu, if not redefined in between
```

Best

```
import modu
[...]
x = modu.sqrt(4)  # sqrt is visibly part of modu's namespace
```

As mentioned in the *Code Style* section, readability is one of the main features of Python. Readability means to avoid useless boilerplate text and clutter, therefore some efforts are spent trying to achieve a certain level of brevity. But terseness and obscurity are the limits where brevity should stop. Being able to tell immediately where a class or function comes from, as in the `modu.func` idiom, greatly improves code readability and understandability in all but the simplest single file projects.

Packages

Python provides a very straightforward packaging system, which is simply an extension of the module mechanism to a directory.

Any directory with an `__init__.py` file is considered a Python package. The different modules in the package are imported in a similar manner as plain modules, but with a special behavior for the `__init__.py` file, which is used to gather all package-wide definitions.

A file `modu.py` in the directory `pack/` is imported with the statement `import pack.modu`. This statement will look for an `__init__.py` file in `pack`, execute all of its top-level statements. Then it will look for a file named

`pack/modu.py` and execute all of its top-level statements. After these operations, any variable, function, or class defined in `modu.py` is available in the pack.modu namespace.

A commonly seen issue is to add too much code to `__init__.py` files. When the project complexity grows, there may be sub-packages and sub-sub-packages in a deep directory structure. In this case, importing a single item from a sub-sub-package will require executing all `__init__.py` files met while traversing the tree.

Leaving an `__init__.py` file empty is considered normal and even a good practice, if the package's modules and sub-packages do not need to share any code.

Lastly, a convenient syntax is available for importing deeply nested packages: `import very.deep.module as mod`. This allows you to use *mod* in place of the verbose repetition of `very.deep.module`.

Object-oriented programming

Python is sometimes described as an object-oriented programming language. This can be somewhat misleading and needs to be clarified.

In Python, everything is an object, and can be handled as such. This is what is meant when we say, for example, that functions are first-class objects. Functions, classes, strings, and even types are objects in Python: like any object, they have a type, they can be passed as function arguments, and they may have methods and properties. In this understanding, Python is an object-oriented language.

However, unlike Java, Python does not impose object-oriented programming as the main programming paradigm. It is perfectly viable for a Python project to not be object-oriented, i.e. to use no or very few class definitions, class inheritance, or any other mechanisms that are specific to object-oriented programming.

Moreover, as seen in the *modules* section, the way Python handles modules and namespaces gives the developer a natural way to ensure the encapsulation and separation of abstraction layers, both being the most common reasons to use object-orientation. Therefore, Python programmers have more latitude to not use object-orientation, when it is not required by the business model.

There are some reasons to avoid unnecessary object-orientation. Defining custom classes is useful when we want to glue together some state and some functionality. The problem, as pointed out by the discussions about functional programming, comes from the "state" part of the equation.

In some architectures, typically web applications, multiple instances of Python processes are spawned to respond to external requests that can happen at the same time. In this case, holding some state into instantiated objects, which means keeping some static information about the world, is prone to concurrency problems or race-conditions. Sometimes, between the initialization of the state of an object (usually done with the `__init__()` method) and the actual use of the object state through one of its methods, the world may have changed, and the retained state may be outdated. For example, a request may load an item in memory and mark it as read by a user. If another request requires the deletion of this item at the same time, it may happen that the deletion actually occurs after the first process loaded the item, and then we have to mark as read a deleted object.

This and other issues led to the idea that using stateless functions is a better programming paradigm.

Another way to say the same thing is to suggest using functions and procedures with as few implicit contexts and side-effects as possible. A function's implicit context is made up of any of the global variables or items in the persistence layer that are accessed from within the function. Side-effects are the changes that a function makes to its implicit context. If a function saves or deletes data in a global variable or in the persistence layer, it is said to have a side-effect.

Carefully isolating functions with context and side-effects from functions with logic (called pure functions) allow the following benefits:

- Pure functions are deterministic: given a fixed input, the output will always be the same.
- Pure functions are much easier to change or replace if they need to be refactored or optimized.

- Pure functions are easier to test with unit-tests: There is less need for complex context setup and data cleaning afterwards.
- Pure functions are easier to manipulate, decorate, and pass around.

In summary, pure functions are more efficient building blocks than classes and objects for some architectures because they have no context or side-effects.

Obviously, object-orientation is useful and even necessary in many cases, for example when developing graphical desktop applications or games, where the things that are manipulated (windows, buttons, avatars, vehicles) have a relatively long life of their own in the computer's memory.

Decorators

The Python language provides a simple yet powerful syntax called 'decorators'. A decorator is a function or a class that wraps (or decorates) a function or a method. The 'decorated' function or method will replace the original 'undecorated' function or method. Because functions are first-class objects in Python, this can be done 'manually', but using the @decorator syntax is clearer and thus preferred.

```
def foo():
    # do something

def decorator(func):
    # manipulate func
    return func

foo = decorator(foo)  # Manually decorate

@decorator
def bar():
    # Do something
# bar() is decorated
```

This mechanism is useful for separating concerns and avoiding external un-related logic 'polluting' the core logic of the function or method. A good example of a piece of functionality that is better handled with decoration is memoization or caching: you want to store the results of an expensive function in a table and use them directly instead of recomputing them when they have already been computed. This is clearly not part of the function logic.

Context Managers

A context manager is a Python object that provides extra contextual information to an action. This extra information takes the form of running a callable upon initiating the context using the `with` statement, as well as running a callable upon completing all the code inside the `with` block. The most well known example of using a context manager is shown here, opening on a file:

```
with open('file.txt') as f:
    contents = f.read()
```

Anyone familiar with this pattern knows that invoking `open` in this fashion ensures that `f`'s `close` method will be called at some point. This reduces a developer's cognitive load and makes the code easier to read.

There are two easy ways to implement this functionality yourself: using a class or using a generator. Let's implement the above functionality ourselves, starting with the class approach:

```
class CustomOpen(object):
    def __init__(self, filename):
      self.file = open(filename)
```

```
    def __enter__(self):
        return self.file

    def __exit__(self, ctx_type, ctx_value, ctx_traceback):
        self.file.close()

with CustomOpen('file') as f:
    contents = f.read()
```

This is just a regular Python object with two extra methods that are used by the `with` statement. CustomOpen is first instantiated and then its `__enter__` method is called and whatever `__enter__` returns is assigned to `f` in the `as f` part of the statement. When the contents of the `with` block is finished executing, the `__exit__` method is then called.

And now the generator approach using Python's own contextlib:

```
from contextlib import contextmanager

@contextmanager
def custom_open(filename):
    f = open(filename)
    try:
        yield f
    finally:
        f.close()

with custom_open('file') as f:
    contents = f.read()
```

This works in exactly the same way as the class example above, albeit it's more terse. The `custom_open` function executes until it reaches the `yield` statement. It then gives control back to the `with` statement, which assigns whatever was `yield`'ed to *f* in the `as f` portion. The `finally` clause ensures that `close()` is called whether or not there was an exception inside the `with`.

Since the two approaches appear the same, we should follow the Zen of Python to decide when to use which. The class approach might be better if there's a considerable amount of logic to encapsulate. The function approach might be better for situations where we're dealing with a simple action.

Dynamic typing

Python is dynamically typed, which means that variables do not have a fixed type. In fact, in Python, variables are very different from what they are in many other languages, specifically statically-typed languages. Variables are not a segment of the computer's memory where some value is written, they are 'tags' or 'names' pointing to objects. It is therefore possible for the variable 'a' to be set to the value 1, then to the value 'a string', then to a function.

The dynamic typing of Python is often considered to be a weakness, and indeed it can lead to complexities and hard-to-debug code. Something named 'a' can be set to many different things, and the developer or the maintainer needs to track this name in the code to make sure it has not been set to a completely unrelated object.

Some guidelines help to avoid this issue:

- Avoid using the same variable name for different things.

Bad

```
a = 1
a = 'a string'
```

```
def a():
    pass  # Do something
```

Good

```
count = 1
msg = 'a string'
def func():
    pass  # Do something
```

Using short functions or methods helps reduce the risk of using the same name for two unrelated things.

It is better to use different names even for things that are related, when they have a different type:

Bad

```
items = 'a b c d'  # This is a string...
items = items.split(' ')  # ...becoming a list
items = set(items)  # ...and then a set
```

There is no efficiency gain when reusing names: the assignments will have to create new objects anyway. However, when the complexity grows and each assignment is separated by other lines of code, including 'if' branches and loops, it becomes harder to ascertain what a given variable's type is.

Some coding practices, like functional programming, recommend never reassigning a variable. In Java this is done with the *final* keyword. Python does not have a *final* keyword and it would be against its philosophy anyway. However, it may be a good discipline to avoid assigning to a variable more than once, and it helps in grasping the concept of mutable and immutable types.

Mutable and immutable types

Python has two kinds of built-in or user-defined types.

Mutable types are those that allow in-place modification of the content. Typical mutables are lists and dictionaries: All lists have mutating methods, like `list.append()` or `list.pop()`, and can be modified in place. The same goes for dictionaries.

Immutable types provide no method for changing their content. For instance, the variable x set to the integer 6 has no "increment" method. If you want to compute x + 1, you have to create another integer and give it a name.

```
my_list = [1, 2, 3]
my_list[0] = 4
print my_list  # [4, 2, 3] <- The same list as changed

x = 6
x = x + 1  # The new x is another object
```

One consequence of this difference in behavior is that mutable types are not "stable", and therefore cannot be used as dictionary keys.

Using properly mutable types for things that are mutable in nature and immutable types for things that are fixed in nature helps to clarify the intent of the code.

For example, the immutable equivalent of a list is the tuple, created with `(1, 2)`. This tuple is a pair that cannot be changed in-place, and can be used as a key for a dictionary.

One peculiarity of Python that can surprise beginners is that strings are immutable. This means that when constructing a string from its parts, it is much more efficient to accumulate the parts in a list, which is mutable, and then glue ('join') the parts together when the full string is needed. One thing to notice, however, is that list comprehensions are better and faster than constructing a list in a loop with calls to `append()`.

Bad

```
# create a concatenated string from 0 to 19 (e.g. "012..1819")
nums = ""
for n in range(20):
  nums += str(n)    # slow and inefficient
print nums
```

Good

```
# create a concatenated string from 0 to 19 (e.g. "012..1819")
nums = []
for n in range(20):
  nums.append(str(n))
print "".join(nums)  # much more efficient
```

Best

```
# create a concatenated string from 0 to 19 (e.g. "012..1819")
nums = [str(n) for n in range(20)]
print "".join(nums)
```

One final thing to mention about strings is that using `join()` is not always best. In the instances where you are creating a new string from a pre-determined number of strings, using the addition operator is actually faster, but in cases like above or in cases where you are adding to an existing string, using `join()` should be your preferred method.

```
foo = 'foo'
bar = 'bar'

foobar = foo + bar  # This is good
foo += 'ooo'  # This is bad, instead you should do:
foo = ''.join([foo, 'ooo'])
```

Note: You can also use the `%` formatting operator to concatenate a pre-determined number of strings besides `str.join()` and +. However, **PEP 3101**, discourages the usage of the `%` operator in favor of the `str.format()` method.

```
foo = 'foo'
bar = 'bar'

foobar = '%s%s' % (foo, bar) # It is OK
foobar = '{0}{1}'.format(foo, bar) # It is better
foobar = '{foo}{bar}'.format(foo=foo, bar=bar) # It is best
```

Vendorizing Dependencies

Runners

Further Reading

- http://docs.python.org/2/library/
- http://www.diveintopython.net/toc/index.html

Code Style

If you ask Python programmers what they like most about Python, they will often cite its high readability. Indeed, a high level of readability is at the heart of the design of the Python language, following the recognized fact that code is read much more often than it is written.

One reason for the high readability of Python code is its relatively complete set of Code Style guidelines and "Pythonic" idioms.

When a veteran Python developer (a Pythonista) calls portions of code not "Pythonic", they usually mean that these lines of code do not follow the common guidelines and fail to express its intent in what is considered the best (hear: most readable) way.

On some border cases, no best way has been agreed upon on how to express an intent in Python code, but these cases are rare.

General concepts

Explicit code

While any kind of black magic is possible with Python, the most explicit and straightforward manner is preferred.

Bad

```
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

Good

```
def make_complex(x, y):
    return {'x': x, 'y': y}
```

In the good code above, x and y are explicitly received from the caller, and an explicit dictionary is returned. The developer using this function knows exactly what to do by reading the first and last lines, which is not the case with the bad example.

One statement per line

While some compound statements such as list comprehensions are allowed and appreciated for their brevity and their expressiveness, it is bad practice to have two disjointed statements on the same line of code.

Bad

```
print 'one'; print 'two'

if x == 1: print 'one'

if <complex comparison> and <other complex comparison>:
    # do something
```

Good

```
print 'one'
print 'two'

if x == 1:
```

```
    print 'one'

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

Function arguments

Arguments can be passed to functions in four different ways.

1. **Positional arguments** are mandatory and have no default values. They are the simplest form of arguments and they can be used for the few function arguments that are fully part of the function's meaning and their order is natural. For instance, in `send(message, recipient)` or `point(x, y)` the user of the function has no difficulty remembering that those two functions require two arguments, and in which order.

In those two cases, it is possible to use argument names when calling the functions and, doing so, it is possible to switch the order of arguments, calling for instance `send(recipient='World', message='Hello')` and `point(y=2, x=1)` but this reduces readability and is unnecessarily verbose, compared to the more straightforward calls to `send('Hello', 'World')` and `point(1, 2)`.

2. **Keyword arguments** are not mandatory and have default values. They are often used for optional parameters sent to the function. When a function has more than two or three positional parameters, its signature is more difficult to remember and using keyword arguments with default values is helpful. For instance, a more complete `send` function could be defined as `send(message, to, cc=None, bcc=None)`. Here `cc` and `bcc` are optional, and evaluate to `None` when they are not passed another value.

Calling a function with keyword arguments can be done in multiple ways in Python, for example it is possible to follow the order of arguments in the definition without explicitly naming the arguments, like in `send('Hello', 'World', 'Cthulhu', 'God')`, sending a blind carbon copy to God. It would also be possible to name arguments in another order, like in `send('Hello again', 'World', bcc='God', cc='Cthulhu')`. Those two possibilities are better avoided without any strong reason to not follow the syntax that is the closest to the function definition: `send('Hello', 'World', cc='Cthulhu', bcc='God')`.

As a side note, following YAGNI principle, it is often harder to remove an optional argument (and its logic inside the function) that was added "just in case" and is seemingly never used, than to add a new optional argument and its logic when needed.

3. The **arbitrary argument list** is the third way to pass arguments to a function. If the function intention is better expressed by a signature with an extensible number of positional arguments, it can be defined with the `*args` constructs. In the function body, `args` will be a tuple of all the remaining positional arguments. For example, `send(message, *args)` can be called with each recipient as an argument:`send('Hello', 'God', 'Mom', 'Cthulhu')`, and in the function body `args` will be equal to `('God', 'Mom', 'Cthulhu')`.

However, this construct has some drawbacks and should be used with caution. If a function receives a list of arguments of the same nature, it is often more clear to define it as a function of one argument, that argument being a list or any sequence. Here, if `send` has multiple recipients, it is better to define it explicitly: `send(message, recipients)` and call it with `send('Hello', ['God', 'Mom', 'Cthulhu'])`. This way, the user of the function can manipulate the recipient list as a list beforehand, and it opens the possibility to pass any sequence, including iterators, that cannot be unpacked as other sequences.

4. The **arbitrary keyword argument dictionary** is the last way to pass arguments to functions. If the function requires an undetermined series of named arguments, it is possible to use the `**kwargs` construct. In the function body, `kwargs` will be a dictionary of all the passed named arguments that have not been caught by other keyword arguments in the function signature.

The same caution as in the case of *arbitrary argument list* is necessary, for similar reasons: these powerful techniques are to be used when there is a proven necessity to use them, and they should not be used if the simpler and clearer construct is sufficient to express the function's intention.

It is up to the programmer writing the function to determine which arguments are positional arguments and which are optional keyword arguments, and to decide whether to use the advanced techniques of arbitrary argument passing. If the advice above is followed wisely, it is possible and enjoyable to write Python functions that are:

- easy to read (the name and arguments need no explanations)
- easy to change (adding a new keyword argument does not break other parts of the code)

Avoid the magical wand

A powerful tool for hackers, Python comes with a very rich set of hooks and tools allowing you to do almost any kind of tricky tricks. For instance, it is possible to do each of the following:

- change how objects are created and instantiated
- change how the Python interpreter imports modules
- it is even possible (and recommended if needed) to embed C routines in Python.

However, all these options have many drawbacks and it is always better to use the most straightforward way to achieve your goal. The main drawback is that readability suffers greatly when using these constructs. Many code analysis tools, such as pylint or pyflakes, will be unable to parse this "magic" code.

We consider that a Python developer should know about these nearly infinite possibilities, because it instills confidence that no impassable problem will be on the way. However, knowing how and particularly when **not** to use them is very important.

Like a kung fu master, a Pythonista knows how to kill with a single finger, and never to actually do it.

We are all responsible users

As seen above, Python allows many tricks, and some of them are potentially dangerous. A good example is that any client code can override an object's properties and methods: there is no "private" keyword in Python. This philosophy, very different from highly defensive languages like Java, which give a lot of mechanisms to prevent any misuse, is expressed by the saying: "We are all responsible users".

This doesn't mean that, for example, no properties are considered private, and that no proper encapsulation is possible in Python. Rather, instead of relying on concrete walls erected by the developers between their code and other's, the Python community prefers to rely on a set of conventions indicating that these elements should not be accessed directly.

The main convention for private properties and implementation details is to prefix all "internals" with an underscore. If the client code breaks this rule and accesses these marked elements, any misbehavior or problems encountered if the code is modified is the responsibility of the client code.

Using this convention generously is encouraged: any method or property that is not intended to be used by client code should be prefixed with an underscore. This will guarantee a better separation of duties and easier modification of existing code; it will always be possible to publicize a private property, but making a public property private might be a much harder operation.

Returning values

When a function grows in complexity it is not uncommon to use multiple return statements inside the function's body. However, in order to keep a clear intent and a sustainable readability level, it is preferable to avoid returning

meaningful values from many output points in the body.

There are two main cases for returning values in a function: the result of the function return when it has been processed normally, and the error cases that indicate a wrong input parameter or any other reason for the function to not be able to complete its computation or task.

If you do not wish to raise exceptions for the second case, then returning a value, such as None or False, indicating that the function could not perform correctly might be needed. In this case, it is better to return as early as the incorrect context has been detected. It will help to flatten the structure of the function: all the code after the return-because-of-error statement can assume the condition is met to further compute the function's main result. Having multiple such return statements is often necessary.

However, when a function has multiple main exit points for its normal course, it becomes difficult to debug the returned result, so it may be preferable to keep a single exit point. This will also help factoring out some code paths, and the multiple exit points are a probable indication that such a refactoring is needed.

```
def complex_function(a, b, c):
    if not a:
        return None  # Raising an exception might be better
    if not b:
        return None  # Raising an exception might be better
    # Some complex code trying to compute x from a, b and c
    # Resist temptation to return x if succeeded
    if not x:
        # Some Plan-B computation of x
    return x  # One single exit point for the returned value x will help
              # when maintaining the code.
```

Idioms

A programming idiom, put simply, is a *way* to write code. The notion of programming idioms is discussed amply at c2 and at Stack Overflow.

Idiomatic Python code is often referred to as being *Pythonic*.

Although there usually is one — and preferably only one — obvious way to do it; *the* way to write idiomatic Python code can be non-obvious to Python beginners. So, good idioms must be consciously acquired.

Some common Python idioms follow:

Unpacking

If you know the length of a list or tuple, you can assign names to its elements with unpacking. For example, since `enumerate()` will provide a tuple of two elements for each item in list:

```
for index, item in enumerate(some_list):
    # do something with index and item
```

You can use this to swap variables as well:

```
a, b = b, a
```

Nested unpacking works too:

```
a, (b, c) = 1, (2, 3)
```

In Python 3, a new method of extended unpacking was introduced by **PEP 3132**:

```
a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]
a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4
```

Create an ignored variable

If you need to assign something (for instance, in *Unpacking*) but will not need that variable, use __:

```
filename = 'foobar.txt'
basename, __, ext = filename.rpartition('.')
```

Note: Many Python style guides recommend the use of a single underscore "_" for throwaway variables rather than the double underscore "__" recommended here. The issue is that "_" is commonly used as an alias for the `gettext()` function, and is also used at the interactive prompt to hold the value of the last operation. Using a double underscore instead is just as clear and almost as convenient, and eliminates the risk of accidentally interfering with either of these other use cases.

Create a length-N list of the same thing

Use the Python list * operator:

```
four_nones = [None] * 4
```

Create a length-N list of lists

Because lists are mutable, the * operator (as above) will create a list of N references to the *same* list, which is not likely what you want. Instead, use a list comprehension:

```
four_lists = [[] for __ in xrange(4)]
```

Note: Use range() instead of xrange() in Python 3

Create a string from a list

A common idiom for creating strings is to use `str.join()` on an empty string.

```
letters = ['s', 'p', 'a', 'm']
word = ''.join(letters)
```

This will set the value of the variable *word* to 'spam'. This idiom can be applied to lists and tuples.

Searching for an item in a collection

Sometimes we need to search through a collection of things. Let's look at two options: lists and sets.

Take the following code for example:

```
s = set(['s', 'p', 'a', 'm'])
l = ['s', 'p', 'a', 'm']

def lookup_set(s):
    return 's' in s

def lookup_list(l):
    return 's' in l
```

Even though both functions look identical, because *lookup_set* is utilizing the fact that sets in Python are hashtables, the lookup performance between the two is very different. To determine whether an item is in a list, Python will have to go through each item until it finds a matching item. This is time consuming, especially for long lists. In a set, on the other hand, the hash of the item will tell Python where in the set to look for a matching item. As a result, the search can be done quickly, even if the set is large. Searching in dictionaries works the same way. For more information see this StackOverflow page. For detailed information on the amount of time various common operations take on each of these data structures, see this page.

Because of these differences in performance, it is often a good idea to use sets or dictionaries instead of lists in cases where:

- The collection will contain a large number of items
- You will be repeatedly searching for items in the collection
- You do not have duplicate items.

For small collections, or collections which you will not frequently be searching through, the additional time and memory required to set up the hashtable will often be greater than the time saved by the improved search speed.

Zen of Python

Also known as **PEP 20**, the guiding principles for Python's design.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

For some examples of good Python style, see these slides from a Python user group.

PEP 8

PEP 8 is the de-facto code style guide for Python. A high quality, easy-to-read version of PEP 8 is also available at pep8.org.

This is highly recommended reading. The entire Python community does their best to adhere to the guidelines laid out within this document. Some project may sway from it from time to time, while others may amend its recommendations.

That being said, conforming your Python code to PEP 8 is generally a good idea and helps make code more consistent when working on projects with other developers. There is a command-line program, pep8, that can check your code for conformance. Install it by running the following command in your terminal:

```
$ pip install pep8
```

Then run it on a file or series of files to get a report of any violations.

```
$ pep8 optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

The program autopep8 can be used to automatically reformat code in the PEP 8 style. Install the program with:

```
$ pip install autopep8
```

Use it to format a file in-place with:

```
$ autopep8 --in-place optparse.py
```

Excluding the `--in-place` flag will cause the program to output the modified code directly to the console for review. The `--aggressive` flag will perform more substantial changes and can be applied multiple times for greater effect.

Conventions

Here are some conventions you should follow to make your code easier to read.

Check if variable equals a constant

You don't need to explicitly compare a value to True, or None, or 0 - you can just add it to the if statement. See Truth Value Testing for a list of what is considered false.

Bad:

```
if attr == True:
    print 'True!'

if attr == None:
    print 'attr is None!'
```

Good:

```
# Just check the value
if attr:
    print 'attr is truthy!'

# or check for the opposite
if not attr:
    print 'attr is falsey!'

# or, since None is considered false, explicitly check for it
if attr is None:
    print 'attr is None!'
```

Access a Dictionary Element

Don't use the `dict.has_key()` method. Instead, use `x in d` syntax, or pass a default argument to `dict.get()`.

Bad:

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print d['hello']    # prints 'world'
else:
    print 'default_value'
```

Good:

```
d = {'hello': 'world'}

print d.get('hello', 'default_value') # prints 'world'
print d.get('thingy', 'default_value') # prints 'default_value'

# Or:
if 'hello' in d:
    print d['hello']
```

Short Ways to Manipulate Lists

List comprehensions provide a powerful, concise way to work with lists. Also, the `map()` and `filter()` functions can perform operations on lists using a different, more concise syntax.

Bad:

```
# Filter elements greater than 4
a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)
```

Good:

```
a = [3, 4, 5]
b = [i for i in a if i > 4]
# Or:
b = filter(lambda x: x > 4, a)
```

Bad:

```
# Add three to all list members.
a = [3, 4, 5]
for i in range(len(a)):
    a[i] += 3
```

Good:

```
a = [3, 4, 5]
a = [i + 3 for i in a]
# Or:
a = map(lambda i: i + 3, a)
```

Use `enumerate()` keep a count of your place in the list.

```
a = [3, 4, 5]
for i, item in enumerate(a):
    print i, item
# prints
# 0 3
# 1 4
# 2 5
```

The `enumerate()` function has better readability than handling a counter manually. Moreover, it is better optimized for iterators.

Read From a File

Use the `with open` syntax to read from files. This will automatically close files for you.

Bad:

```
f = open('file.txt')
a = f.read()
print a
f.close()
```

Good:

```
with open('file.txt') as f:
    for line in f:
        print line
```

The `with` statement is better because it will ensure you always close the file, even if an exception is raised inside the `with` block.

Line Continuations

When a logical line of code is longer than the accepted limit, you need to split it over multiple physical lines. The Python interpreter will join consecutive lines if the last character of the line is a backslash. This is helpful in some cases, but should usually be avoided because of its fragility: a white space added to the end of the line, after the backslash, will break the code and may have unexpected results.

A better solution is to use parentheses around your elements. Left with an unclosed parenthesis on an end-of-line the Python interpreter will join the next line until the parentheses are closed. The same behavior holds for curly and square braces.

Bad:

```
my_very_big_string = """For a long time I used to go to bed early. Sometimes, \
    when I had put out my candle, my eyes would close so quickly that I had not even \
    time to say “I’m going to sleep.”"""

from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
    yet_another_nice_function
```

Good:

```
my_very_big_string = (
    "For a long time I used to go to bed early. Sometimes, "
    "when I had put out my candle, my eyes would close so quickly "
    "that I had not even time to say “I’m going to sleep.”"
)

from some.deep.module.inside.a.module import (
    a_nice_function, another_nice_function, yet_another_nice_function)
```

However, more often than not, having to split a long logical line is a sign that you are trying to do too many things at the same time, which may hinder readability.

Reading Great Code

One of the core tenets behind the design of Python is creating readable code. The motivation behind this design is simple: The number one thing that Python programmers do is read code.

One of the secrets of becoming a great Python programmer is to read, understand, and comprehend excellent code.

Excellent code typically follows the guidelines outlined in *Code Style*, and does its best to express a clear and concise intent to the reader.

Included below is a list of recommended Python projects for reading. Each one of these projects is a paragon of Python coding.

- Howdoi Howdoi is a code search tool, written in Python.
- Flask Flask is a microframework for Python based on Werkzeug and Jinja2. It’s intended for getting started very quickly and was developed with best intentions in mind.
- Diamond Diamond is a python daemon that collects metrics and publishes them to Graphite or other backends. It is capable of collecting cpu, memory, network, i/o, load and disk metrics. Additionally, it features an API for implementing custom collectors for gathering metrics from almost any source.
- Werkzeug Werkzeug started as simple collection of various utilities for WSGI applications and has become one of the most advanced WSGI utility modules. It includes a powerful debugger, full-featured request and response objects, HTTP utilities to handle entity tags, cache control headers, HTTP dates, cookie handling, file uploads, a powerful URL routing system and a bunch of community-contributed addon modules.
- Requests Requests is an Apache2 Licensed HTTP library, written in Python, for human beings.
- Tablib Tablib is a format-agnostic tabular dataset library, written in Python.

Todo

Include code examples of exemplary code from each of the projects listed. Explain why it is excellent code. Use complex examples.

Todo

Explain techniques to rapidly identify data structures, algorithms and determine what the code is doing.

Documentation

Readability is a primary focus for Python developers, in both project and code documentation. Following some simple best practices can save both you and others a lot of time.

Project Documentation

A `README` file at the root directory should give general information to both users and maintainers of a project. It should be raw text or written in some very easy to read markup, such as *reStructuredText* or Markdown. It should contain a few lines explaining the purpose of the project or library (without assuming the user knows anything about the project), the URL of the main source for the software, and some basic credit information. This file is the main entry point for readers of the code.

An `INSTALL` file is less necessary with Python. The installation instructions are often reduced to one command, such as `pip install module` or `python setup.py install` and added to the `README` file.

A `LICENSE` file should *always* be present and specify the license under which the software is made available to the public.

A `TODO` file or a `TODO` section in `README` should list the planned development for the code.

A `CHANGELOG` file or section in `README` should compile a short overview of the changes in the code base for the latest versions.

Project Publication

Depending on the project, your documentation might include some or all of the following components:

- An *introduction* should show a very short overview of what can be done with the product, using one or two extremely simplified use cases. This is the thirty-second pitch for your project.
- A *tutorial* should show some primary use cases in more detail. The reader will follow a step-by-step procedure to set-up a working prototype.
- An *API reference* is typically generated from the code (see *docstrings*). It will list all publicly available interfaces, parameters, and return values.
- *Developer documentation* is intended for potential contributors. This can include code convention and general design strategy of the project.

Sphinx

Sphinx is far and away the most popular Python documentation tool. **Use it.** It converts *reStructuredText* markup language into a range of output formats including HTML, LaTeX (for printable PDF versions), manual pages, and plain text.

There is also **great**, **free** hosting for your Sphinx docs: Read The Docs. Use it. You can configure it with commit hooks to your source repository so that rebuilding your documentation will happen automatically.

When run, Sphinx will import your code and using Python's introspection features it will extract all function, method and class signatures. It will also extract the accompanying docstrings, and compile it all into well structured and easily readable documentation for your project.

Note: Sphinx is famous for its API generation, but it also works well for general project documentation. This Guide is built with Sphinx and is hosted on Read The Docs

---

reStructuredText

Most Python documentation is written with reStructuredText. It's like Markdown with all the optional extensions built in.

The reStructuredText Primer and the reStructuredText Quick Reference should help you familiarize yourself with its syntax.

Code Documentation Advice

Comments clarify the code and they are added with purpose of making the code easier to understand. In Python, comments begin with a hash (number sign) (#). In Python, *docstrings* describe modules, classes, and functions:

```
def square_and_rooter(x):
    """Return the square root of self times self."""
    ...
```

In general, follow the comment section of **PEP 8#comments** (the "Python Style Guide"). More information about docstrings can be found at **PEP 0257#specification** (The Docstring Conventions Guide).

Commenting Sections of Code

Do not use triple-quote strings to comment code. This is not a good practice, because line-oriented command-line tools such as grep will not be aware that the commented code is inactive. It is better to add hashes at the proper indentation level for every commented line. Your editor probably has the ability to do this easily, and it is worth learning the comment/uncomment toggle.

Docstrings and Magic

Some tools use docstrings to embed more-than-documentation behavior, such as unit test logic. Those can be nice, but you won't ever go wrong with vanilla "here's what this does."

Tools like Sphinx will parse your docstrings as reStructuredText and render it correctly as HTML. This makes it very easy to embed snippets of example code in a project's documentation.

Additionally, Doctest will read all embedded docstrings that look like input from the Python commandline (prefixed with ">>>") and run them, checking to see if the output of the command matches the text on the following line. This allows developers to embed real examples and usage of functions alongside their source code, and as a side effect, it also ensures that their code is tested and works.

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
```

```
    'aaa'
    """
    return a * b
```

Docstrings versus Block comments

These aren't interchangeable. For a function or class, the leading comment block is a programmer's note. The docstring describes the *operation* of the function or class:

```
# This function slows down program execution for some reason.
def square_and_rooter(x):
    """Returns the square root of self times self."""
    ...
```

Unlike block comments, docstrings are built into the Python language itself. This means you can use all of Python's powerful introspection capabilities to access docstrings at runtime, compared with comments which are optimised out. Docstrings are accessible from both the *__doc__* dunder attribute for almost every Python object, as well as with the built in *help()* function.

While block comments are usually used to explain *what* a section of code is doing, or the specifics of an algorithm, docstrings are more intended for explaining to other users of your code (or you in 6 months time) *how* a particular function can be used and the general purpose of a function, class, or module.

Writing Docstrings

Depending on the complexity of the function, method, or class being written, a one-line docstring may be perfectly appropriate. These are generally used for really obvious cases, such as:

```
def add(a, b):
    """Add two numbers and return the result."""
    return a + b
```

The docstring should describe the function in a way that is easy to understand. For simple cases like trivial functions and classes, simply embedding the function's signature (i.e. *add(a, b) -> result*) in the docstring is unnecessary. This is because with Python's *inspect* module, it is already quite easy to find this information if needed, and it is also readily available by reading the source code.

In larger or more complex projects however, it is often a good idea to give more information about a function, what it does, any exceptions it may raise, what it returns, or relevant details about the parameters.

For more detailed documentation of code a popular style is the one used for the Numpy project, often called Numpy style docstrings. While it can take up a few more lines the previous example, it allows the developer to include a lot more information about a method, function, or class.

```
def random_number_generator(arg1, arg2):
    """
    Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2
```

```
    Returns
    -------
    int
        Description of return value

    """
    return 42
```

The sphinx.ext.napoleon plugin allows Sphinx to parse this style of docstrings, making it easy to incorporate NumPy style docstrings into your project.

At the end of the day, it doesn't really matter what style is used for writing docstrings, their purpose is to serve as documentation for anyone who may need to read or make changes to your code. As long as it is correct, understandable and gets the relevant points across then it has done the job it was designed to do.

For further reading on docstrings, feel free to consult **PEP 257**

Other Tools

You might see these in the wild. Use *Sphinx*.

Pycco Pycco is a "literate-programming-style documentation generator" and is a port of the node.js Docco. It makes code into a side-by-side HTML code and documentation.

Ronn Ronn builds Unix manuals. It converts human readable textfiles to roff for terminal display, and also to HTML for the web.

Epydoc Epydoc is discontinued. Use *Sphinx* instead.

MkDocs MkDocs is a fast and simple static site generator that's geared towards building project documentation with Markdown.

Testing Your Code

Testing your code is very important.

Getting used to writing testing code and running this code in parallel is now considered a good habit. Used wisely, this method helps you define more precisely your code's intent and have a more decoupled architecture.

Some general rules of testing:

- A testing unit should focus on one tiny bit of functionality and prove it correct.
- Each test unit must be fully independent. Each test must be able to run alone, and also within the test suite, regardless of the order that they are called. The implication of this rule is that each test must be loaded with a fresh dataset and may have to do some cleanup afterwards. This is usually handled by `setUp()` and `tearDown()` methods.
- Try hard to make tests that run fast. If one single test needs more than a few milliseconds to run, development will be slowed down or the tests will not be run as often as is desirable. In some cases, tests can't be fast because they need a complex data structure to work on, and this data structure must be loaded every time the test runs. Keep these heavier tests in a separate test suite that is run by some scheduled task, and run all other tests as often as needed.
- Learn your tools and learn how to run a single test or a test case. Then, when developing a function inside a module, run this function's tests frequently, ideally automatically when you save the code.

- Always run the full test suite before a coding session, and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
- It is a good idea to implement a hook that runs all tests before pushing code to a shared repository.
- If you are in the middle of a development session and have to interrupt your work, it is a good idea to write a broken unit test about what you want to develop next. When coming back to work, you will have a pointer to where you were and get back on track faster.
- The first step when you are debugging your code is to write a new test pinpointing the bug. While it is not always possible to do, those bug catching tests are among the most valuable pieces of code in your project.
- Use long and descriptive names for testing functions. The style guide here is slightly different than that of running code, where short names are often preferred. The reason is testing functions are never called explicitly. `square()` or even `sqr()` is ok in running code, but in testing code you would have names such as `test_square_of_number_2()`, `test_square_negative_number()`. These function names are displayed when a test fails, and should be as descriptive as possible.
- When something goes wrong or has to be changed, and if your code has a good set of tests, you or other maintainers will rely largely on the testing suite to fix the problem or modify a given behavior. Therefore the testing code will be read as much as or even more than the running code. A unit test whose purpose is unclear is not very helpful in this case.
- Another use of the testing code is as an introduction to new developers. When someone will have to work on the code base, running and reading the related testing code is often the best thing that they can do to start. They will or should discover the hot spots, where most difficulties arise, and the corner cases. If they have to add some functionality, the first step should be to add a test to ensure that the new functionality is not already a working path that has not been plugged into the interface.

The Basics

Unittest

`unittest` is the batteries-included test module in the Python standard library. Its API will be familiar to anyone who has used any of the JUnit/nUnit/CppUnit series of tools.

Creating test cases is accomplished by subclassing `unittest.TestCase`.

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
```

As of Python 2.7 unittest also includes its own test discovery mechanisms.

> unittest in the standard library documentation

Doctest

The `doctest` module searches for pieces of text that look like interactive Python sessions in docstrings, and then executes those sessions to verify that they work exactly as shown.

Doctests have a different use case than proper unit tests: they are usually less detailed and don't catch special cases or obscure regression bugs. They are useful as an expressive documentation of the main use cases of a module and its components. However, doctests should run automatically each time the full test suite runs.

A simple doctest in a function:

```
def square(x):
    """Return the square of x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """

    return x * x

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

When running this module from the command line as in `python module.py`, the doctests will run and complain if anything is not behaving as described in the docstrings.

Tools

py.test

py.test is a no-boilerplate alternative to Python's standard unittest module.

```
$ pip install pytest
```

Despite being a fully-featured and extensible test tool, it boasts a simple syntax. Creating a test suite is as easy as writing a module with a couple of functions:

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

and then running the *py.test* command

```
$ py.test
========================== test session starts ==========================
platform darwin -- Python 2.7.1 -- pytest-2.2.1
collecting ... collected 1 items

test_sample.py F

=================================== FAILURES ===================================
_____________________________________ test_answer _____________________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)
```

```
test_sample.py:5: AssertionError
========================== 1 failed in 0.02 seconds ===========================
```

is far less work than would be required for the equivalent functionality with the unittest module!

py.test

Nose

nose extends unittest to make testing easier.

```
$ pip install nose
```

nose provides automatic test discovery to save you the hassle of manually creating test suites. It also provides numerous plugins for features such as xUnit-compatible test output, coverage reporting, and test selection.

nose

tox

tox is a tool for automating test environment management and testing against multiple interpreter configurations

```
$ pip install tox
```

tox allows you to configure complicated multi-parameter test matrices via a simple ini-style configuration file.

tox

Unittest2

unittest2 is a backport of Python 2.7's unittest module which has an improved API and better assertions over the one available in previous versions of Python.

If you're using Python 2.6 or below, you can install it with pip

```
$ pip install unittest2
```

You may want to import the module under the name unittest to make porting code to newer versions of the module easier in the future

```
import unittest2 as unittest

class MyTest(unittest.TestCase):
    ...
```

This way if you ever switch to a newer Python version and no longer need the unittest2 module, you can simply change the import in your test module without the need to change any other code.

unittest2

mock

`unittest.mock` is a library for testing in Python. As of Python 3.3, it is available in the standard library.

For older versions of Python:

```
$ pip install mock
```

It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

For example, you can monkey-patch a method:

```
from mock import MagicMock
thing = ProductionClass()
thing.method = MagicMock(return_value=3)
thing.method(3, 4, 5, key='value')

thing.method.assert_called_with(3, 4, 5, key='value')
```

To mock classes or objects in a module under test, use the `patch` decorator. In the example below, an external search system is replaced with a mock that always returns the same result (but only for the duration of the test).

```
def mock_search(self):
    class MockSearchQuerySet(SearchQuerySet):
        def __iter__(self):
            return iter(["foo", "bar", "baz"])
    return MockSearchQuerySet()

# SearchForm here refers to the imported class reference in myapp,
# not where the SearchForm class itself is imported from
@mock.patch('myapp.SearchForm.search', mock_search)
def test_new_watchlist_activities(self):
    # get_search_results runs a search and iterates over the result
    self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

Mock has many other ways you can configure it and control its behavior.

mock

Logging

The `logging` module has been a part of Python's Standard Library since version 2.3. It is succinctly described in **PEP 282**. The documentation is notoriously hard to read, except for the basic logging tutorial.

Logging serves two purposes:

- **Diagnostic logging** records events related to the application's operation. If a user calls in to report an error, for example, the logs can be searched for context.
- **Audit logging** records events for business analysis. A user's transactions can be extracted and combined with other user details for reports or to optimize a business goal.

... or Print?

The only time that `print` is a better option than logging is when the goal is to display a help statement for a command line application. Other reasons why logging is better than `print`:

- The log record, which is created with every logging event, contains readily available diagnostic information such as the file name, full path, function, and line number of the logging event.
- Events logged in included modules are automatically accessible via the root logger to your application's logging stream, unless you filter them out.

- Logging can be selectively silenced by using the method `logging.Logger.setLevel()` or disabled by setting the attribute `logging.Logger.disabled` to `True`.

Logging in a Library

Notes for configuring logging for a library are in the logging tutorial. Because the *user*, not the library, should dictate what happens when a logging event occurs, one admonition bears repeating:

Note: It is strongly advised that you do not add any handlers other than NullHandler to your library's loggers.

Best practice when instantiating loggers in a library is to only create them using the `__name__` global variable: the `logging` module creates a hierarchy of loggers using dot notation, so using `__name__` ensures no name collisions.

Here is an example of best practice from the requests source – place this in your `__init__.py`

```
# Set default logging handler to avoid "No handler found" warnings.
import logging
try:  # Python 2.7+
    from logging import NullHandler
except ImportError:
    class NullHandler(logging.Handler):
        def emit(self, record):
            pass

logging.getLogger(__name__).addHandler(NullHandler())
```

Logging in an Application

The twelve factor app, an authoritative reference for good practice in application development, contains a section on logging best practice. It emphatically advocates for treating log events as an event stream, and for sending that event stream to standard output to be handled by the application environment.

There are at least three ways to configure a logger:

- **Using an INI-formatted file:**
  - **Pro**: possible to update configuration while running using the function `logging.config.listen()` to listen on a socket.
  - **Con**: less control (*e.g.* custom subclassed filters or loggers) than possible when configuring a logger in code.
- **Using a dictionary or a JSON-formatted file:**
  - **Pro**: in addition to updating while running, it is possible to load from a file using the `json` module, in the standard library since Python 2.6.
  - **Con**: less control than when configuring a logger in code.
- **Using code:**
  - **Pro**: complete control over the configuration.
  - **Con**: modifications require a change to source code.

Example Configuration via an INI File

Let us say the file is named `logging_config.ini`. More details for the file format are in the logging configuration section of the logging tutorial.

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Then use `logging.config.fileConfig()` in the code:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Example Configuration via a Dictionary

As of Python 2.7, you can use a dictionary with configuration details. **PEP 391** contains a list of the mandatory and optional elements in the configuration dictionary.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
        },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
```

```
        },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Example Configuration Directly in Code

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Common Gotchas

For the most part, Python aims to be a clean and consistent language that avoids surprises. However, there are a few cases that can be confusing to newcomers.

Some of these cases are intentional but can be potentially surprising. Some could arguably be considered language warts. In general, what follows is a collection of potentially tricky behavior that might seem strange at first glance, but is generally sensible once you're aware of the underlying cause for the surprise.

Mutable Default Arguments

Seemingly the *most* common surprise new Python programmers encounter is Python's treatment of mutable default arguments in function definitions.

What You Wrote

```
def append_to(element, to=[]):
    to.append(element)
    return to
```

What You Might Have Expected to Happen

```
my_list = append_to(12)
print my_list

my_other_list = append_to(42)
print my_other_list
```

A new list is created each time the function is called if a second argument isn't provided, so that the output is:

```
[12]
[42]
```

What Does Happen

```
[12]
[12, 42]
```

A new list is created *once* when the function is defined, and the same list is used in each successive call.

Python's default arguments are evaluated *once* when the function is defined, not each time the function is called (like it is in say, Ruby). This means that if you use a mutable default argument and mutate it, you *will* and have mutated that object for all future calls to the function as well.

What You Should Do Instead

Create a new object each time the function is called, by using a default arg to signal that no argument was provided (`None` is often a good choice).

```
def append_to(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

When the Gotcha Isn't a Gotcha

Sometimes you can specifically "exploit" (read: use as intended) this behavior to maintain state between calls of a function. This is often done when writing a caching function.

Late Binding Closures

Another common source of confusion is the way Python binds its variables in closures (or in the surrounding global scope).

What You Wrote

```
def create_multipliers():
    return [lambda x : i * x for i in range(5)]
```

What You Might Have Expected to Happen

```
for multiplier in create_multipliers():
    print multiplier(2)
```

A list containing five functions that each have their own closed-over `i` variable that multiplies their argument, producing:

```
0
2
4
6
8
```

What Does Happen

```
8
8
8
8
8
```

Five functions are created; instead all of them just multiply `x` by 4.

Python's closures are *late binding*. This means that the values of variables used in closures are looked up at the time the inner function is called.

Here, whenever *any* of the returned functions are called, the value of `i` is looked up in the surrounding scope at call time. By then, the loop has completed and `i` is left with its final value of 4.

What's particularly nasty about this gotcha is the seemingly prevalent misinformation that this has something to do with lambdas in Python. Functions created with a `lambda` expression are in no way special, and in fact the same exact behavior is exhibited by just using an ordinary `def`:

```
def create_multipliers():
    multipliers = []

    for i in range(5):
        def multiplier(x):
            return i * x
        multipliers.append(multiplier)

    return multipliers
```

What You Should Do Instead

The most general solution is arguably a bit of a hack. Due to Python's aforementioned behavior concerning evaluating default arguments to functions (see *Mutable Default Arguments*), you can create a closure that binds immediately to its arguments by using a default arg like so:

```
def create_multipliers():
    return [lambda x, i=i : i * x for i in range(5)]
```

Alternatively, you can use the functools.partial function:

```
from functools import partial
from operator import mul

def create_multipliers():
    return [partial(mul, i) for i in range(5)]
```

When the Gotcha Isn't a Gotcha

Sometimes you want your closures to behave this way. Late binding is good in lots of situations. Looping to create unique functions is unfortunately a case where they can cause hiccups.

Bytecode (.pyc) Files Everywhere!

By default, when executing Python code from files, the Python interpreter will automatically write a bytecode version of that file to disk, e.g. `module.pyc`.

These `.pyc` files should not be checked into your source code repositories.

Theoretically, this behavior is on by default, for performance reasons. Without these bytecode files present, Python would re-generate the bytecode every time the file is loaded.

Disabling Bytecode (.pyc) Files

Luckily, the process of generating the bytecode is extremely fast, and isn't something you need to worry about while developing your code.

Those files are annoying, so let's get rid of them!

```
$ export PYTHONDONTWRITEBYTECODE=1
```

With the `$PYTHONDONTWRITEBYTECODE` environment variable set, Python will no longer write these files to disk, and your development environment will remain nice and clean.

I recommend setting this environment variable in your `~/.profile`.

Removing Bytecode (.pyc) Files

Here's nice trick for removing all of these files, if they already exist:

```
$ find . -type f -name "*.py[co]" -delete -or -type d -name "__pycache__" -delete
```

Run that from the root directory of your project, and all `.pyc` files will suddenly vanish. Much better.

Choosing a License

Your source publication *needs* a license. In the US, if no license is specified, users have no legal right to download, modify, or distribute. Furthermore, people can't contribute to your code unless you tell them what rules to play by. Choosing a license is complicated, so here are some pointers:

Open source. There are plenty of open source licenses available to choose from.

In general, these licenses tend to fall into one of two categories:

1. licenses that focus more on the user's freedom to do with the software as they please (these are the more permissive open source licenses such as the MIT, BSD, & Apache).
2. licenses that focus more on making sure that the code itself — including any changes made to it and distributed along with it — always remains free (these are the less permissive free software licenses such as the GPL and LGPL).

The latter are less permissive in the sense that they don't permit someone to add code to the software and distribute it without also including the source code for their changes.

To help you choose one for your project, there's a license chooser, **use it**.

More Permissive

- PSFL (Python Software Foundation License) – for contributing to Python itself
- MIT / BSD / ISC
  - MIT (X11)
  - New BSD
  - ISC
- Apache

Less Permissive:

- LGPL
- GPL
  - GPLv2
  - GPLv3

A good overview of licenses with explanations of what one can, cannot, and must do using a particular software can be found at tl;drLegal.

CHAPTER 3

Scenario Guide for Python Applications

This part of the guide focuses on tool and module advice based on different scenarios.

Network Applications

HTTP

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

Requests

Python's standard urllib2 module provides most of the HTTP capabilities you need, but the API is thoroughly broken. It was built for a different time — and a different web. It requires an enormous amount of work (even method overrides) to perform the simplest of tasks.

Requests takes all of the work out of Python HTTP — making your integration with web services seamless. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, powered by urllib3, which is embedded within Requests.

- Documentation
- PyPi
- GitHub

Distributed Systems

ZeroMQ

ØMQ (also spelled ZeroMQ, 0MQ or ZMQ) is a high-performance asynchronous messaging library aimed at use in scalable distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ØMQ system can run without a dedicated message broker. The library is designed to have a familiar socket-style API.

RabbitMQ

RabbitMQ is an open source message broker software that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover. Client libraries to interface with the broker are available for all major programming languages.

- Homepage
- GitHub Organization

Web Applications & Frameworks

As a powerful scripting language adapted to both fast prototyping and bigger projects, Python is widely used in web application development.

Context

WSGI

The Web Server Gateway Interface (or "WSGI" for short) is a standard interface between web servers and Python web application frameworks. By standardizing behavior and communication between web servers and Python web frameworks, WSGI makes it possible to write portable Python web code that can be deployed in any *WSGI-compliant web server*. WSGI is documented in **PEP 3333**.

Frameworks

Broadly speaking, a web framework consists of a set of libraries and a main handler within which you can build custom code to implement a web application (i.e. an interactive web site). Most web frameworks include patterns and utilities to accomplish at least the following:

URL Routing Matches an incoming HTTP request to a particular piece of Python code to be invoked

Request and Response Objects Encapsulate the information received from or sent to a user's browser

Template Engine Allows for separating Python code implementing an application's logic from the HTML (or other) output that it produces

Development Web Server Runs an HTTP server on development machines to enable rapid development; often automatically reloads server-side code when files are updated

Django

Django is a "batteries included" web application framework, and is an excellent choice for creating content-oriented websites. By providing many utilities and patterns out of the box, Django aims to make it possible to build complex, database-backed web applications quickly, while encouraging best practices in code written using it.

Django has a large and active community, and many pre-built re-usable modules that can be incorporated into a new project as-is, or customized to fit your needs.

There are annual Django conferences in the United States and in Europe.

The majority of new Python web applications today are built with Django.

Flask

Flask is a "microframework" for Python, and is an excellent choice for building smaller applications, APIs, and web services.

Building an app with Flask is a lot like writing standard Python modules, except some functions have routes attached to them. It's really beautiful.

Rather than aiming to provide everything you could possibly need, Flask implements the most commonly-used core components of a web application framework, like URL routing, request and response objects, and templates.

If you use Flask, it is up to you to choose other components for your application, if any. For example, database access or form generation and validation are not built-in functions of Flask.

This is great, because many web applications don't need those features. For those that do, there are many Extensions available that may suit your needs. Or, you can easily use any library you want yourself!

Flask is default choice for any Python web application that isn't a good fit for Django.

Tornado

Tornado is an asyncronous web framework for Python that has its own event loop. This allows it to natively support WebSockets, for example. Well-written Tornado applications are known to have excellent performance characteristics.

I do not recommend using Tornado unless you think you need it.

Pyramid

Pyramid is a very flexible framework with a heavy focus on modularity. It comes with a small number of libraries ("batteries") built-in, and encourages users to extend its base functionality.

Pyramid does not have a large user base, unlike Django and Flask. It's a capable framework, but not a very popular choice for new Python web applications today.

Web Servers

Nginx

Nginx (pronounced "engine-x") is a web server and reverse-proxy for HTTP, SMTP and other protocols. It is known for its high performance, relative simplicity, and compatibility with many application servers (like WSGI servers). It also includes handy features like load-balancing, basic authentication, streaming, and others. Designed to serve high-load websites, Nginx is gradually becoming quite popular.

WSGI Servers

Stand-alone WSGI servers typically use less resources than traditional web servers and provide top performance [3].

[3] Benchmark of Python WSGI Servers

Gunicorn

Gunicorn (Green Unicorn) is a pure-python WSGI server used to serve Python applications. Unlike other Python web servers, it has a thoughtful user-interface, and is extremely easy to use and configure.

Gunicorn has sane and reasonable defaults for configurations. However, some other servers, like uWSGI, are tremendously more customizable, and therefore, are much more difficult to effectively use.

Gunicorn is the recommended choice for new Python web applications today.

Waitress

Waitress is a pure-python WSGI server that claims "very acceptable performance". Its documentation is not very detailed, but it does offer some nice functionality that Gunicorn doesn't have (e.g. HTTP request buffering).

Waitress is gaining popularity within the Python web development community.

uWSGI

uWSGI is a full stack for building hosting services. In addition to process management, process monitoring, and other functionality, uWSGI acts as an application server for various programming languages and protocols - including Python and WSGI. uWSGI can either be run as a stand-alone web router, or be run behind a full web server (such as Nginx or Apache). In the latter case, a web server can configure uWSGI and an application's operation over the uwsgi protocol. uWSGI's web server support allows for dynamically configuring Python, passing environment variables and further tuning. For full details, see uWSGI magic variables.

I do not recommend using uWSGI unless you know why you need it.

Server Best Practices

The majority of self-hosted Python applications today are hosted with a WSGI server such as *Gunicorn*, either directly or behind a lightweight web server such as *nginx*.

The WSGI servers serve the Python applications while the web server handles tasks better suited for it such as static file serving, request routing, DDoS protection, and basic authentication.

Hosting

Platform-as-a-Service (PaaS) is a type of cloud computing infrastructure which abstracts and manages infrastructure, routing, and scaling of web applications. When using a PaaS, application developers can focus on writing application code rather than needing to be concerned with deployment details.

Heroku

Heroku offers first-class support for Python 2.7–3.5 applications.

Heroku supports all types of Python web applications, servers, and frameworks. Applications can be developed on Heroku for free. Once your application is ready for production, you can upgrade to a Hobby or Professional application.

Heroku maintains detailed articles on using Python with Heroku, as well as step-by-step instructions on how to set up your first application.

Heroku is the recommended PaaS for deploying Python web applications today.

Eldarion

Eldarion (formely known as Gondor) is a PaaS powered by Kubernetes, CoreOS, and Docker. They support any WSGI application and have a guide on deploying Django projects.

Templating

Most WSGI applications are responding to HTTP requests to serve content in HTML or other markup languages. Instead of generating directly textual content from Python, the concept of separation of concerns advises us to use templates. A template engine manages a suite of template files, with a system of hierarchy and inclusion to avoid unnecessary repetition, and is in charge of rendering (generating) the actual content, filling the static content of the templates with the dynamic content generated by the application.

As template files are sometimes written by designers or front-end developers, it can be difficult to handle increasing complexity.

Some general good practices apply to the part of the application passing dynamic content to the template engine, and to the templates themselves.

- Template files should be passed only the dynamic content that is needed for rendering the template. Avoid the temptation to pass additional content "just in case": it is easier to add some missing variable when needed than to remove a likely unused variable later.
- Many template engines allow for complex statements or assignments in the template itself, and many allow some Python code to be evaluated in the templates. This convenience can lead to uncontrolled increase in complexity, and often make it harder to find bugs.
- It is often necessary to mix JavaScript templates with HTML templates. A sane approach to this design is to isolate the parts where the HTML template passes some variable content to the JavaScript code.

Jinja2

Jinja2 is a very well-regarded template engine.

It uses a text-based template language and can thus be used to generate any type markup, not just HTML. It allows customization of filters, tags, tests and globals. It features many improvements over Django's templating system.

Here some important html tags in Jinja2:

```
{# This is a comment #}

{# The next tag is a variable output: #}
{{title}}

{# Tag for a block, can be replaced through inheritance with other html code #}
{% block head %}
<h1>This is the head!</h1>
{% endblock %}

{# Output of an array as an iteration #}
{% for item in list %}
<li>{{ item }}</li>
{% endfor %}
```

The next listings is an example of a web site in combination with the Tornado web server. Tornado is not very complicated to use.

```
# import Jinja2
from jinja2 import Environment, FileSystemLoader

# import Tornado
import tornado.ioloop
import tornado.web

# Load template file templates/site.html
TEMPLATE_FILE = "site.html"
templateLoader = FileSystemLoader( searchpath="templates/" )
templateEnv = Environment( loader=templateLoader )
template = templateEnv.get_template(TEMPLATE_FILE)

# List for famous movie rendering
movie_list = [[1,"The Hitchhiker's Guide to the Galaxy"],[2,"Back to future"],[3,"Matrix"]]

# template.render() returns a string which contains the rendered html
html_output = template.render(list=movie_list,
                        title="Here is my favorite movie list")

# Handler for main page
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        # Returns rendered template string to the browser request
        self.write(html_output)

# Assign handler to the server root  (127.0.0.1:PORT/)
application = tornado.web.Application([
    (r"/", MainHandler),
])
PORT=8884
if __name__ == "__main__":
    # Setup the server
    application.listen(PORT)
    tornado.ioloop.IOLoop.instance().start()
```

The `base.html` file can be used as base for all site pages which are for example implemented in the content block.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{{title}} - My Webpage</title>
</head>
<body>
<div id="content">
    {# In the next line the content from the site.html template will be added #}
    {% block content %}{% endblock %}
</div>
<div id="footer">
    {% block footer %}
    &copy; Copyright 2013 by <a href="http://domain.invalid/">you</a>.
    {% endblock %}
</div>
</body>
```

The next listing is our site page (`site.html`) loaded in the Python app which extends `base.html`. The content block is automatically set into the corresponding block in the `base.html` page.

```
<!{% extends "base.html" %}
{% block content %}
    <p class="important">
    <div id="content">
        <h2>{{title}}</h2>
        <p>{{ list_title }}</p>
        <ul>
             {% for item in list %}
             <li>{{ item[0]}} :  {{ item[1]}}</li>
             {% endfor %}
        </ul>
    </div>
    </p>
{% endblock %}
```

Jinja2 is the recommended templating library for new Python web applications.

Chameleon

Chameleon Page Templates are an HTML/XML template engine implementation of the Template Attribute Language (TAL), TAL Expression Syntax (TALES), and Macro Expansion TAL (Metal) syntaxes.

Chameleon is available for Python 2.5 and up (including 3.x and pypy), and is commonly used by the Pyramid Framework.

Page Templates add within your document structure special element attributes and text markup. Using a set of simple language constructs, you control the document flow, element repetition, text replacement and translation. Because of the attribute-based syntax, unrendered page templates are valid HTML and can be viewed in a browser and even edited in WYSIWYG editors. This can make round-trip collaboration with designers and prototyping with static files in a browser easier.

The basic TAL language is simple enough to grasp from an example:

```
<html>
  <body>
  <h1>Hello, <span tal:replace="context.name">World</span>!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
           <span tal:replace="row.capitalize()" /> <span tal:replace="col" />
        </td>
      </tr>
    </table>
  </body>
</html>
```

The ** pattern for text insertion is common enough that if you do not require strict validity in your unrendered templates, you can replace it with a more terse and readable syntax that uses the pattern *${expression}*, as follows:

```
<html>
  <body>
    <h1>Hello, ${world}!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
           ${row.capitalize()} ${col}
        </td>
```

```
      </tr>
    </table>
  </body>
</html>
```

But keep in mind that the full *Default Text* syntax also allows for default content in the unrendered template.

Being from the Pyramid world, Chameleon is not widely used.

Mako

Mako is a template language that compiles to Python for maximum performance. Its syntax and api is borrowed from the best parts of other templating languages like Django and Jinja2 templates. It is the default template language included with the Pylons and Pyramid web frameworks.

An example template in Mako looks like:

```
<%inherit file="base.html"/>
<%
    rows = [[v for v in range(0,10)] for row in range(0,10)]
%>
<table>
    % for row in rows:
        ${makerow(row)}
    % endfor
</table>

<%def name="makerow(row)">
    <tr>
    % for name in row:
        <td>${name}</td>\
    % endfor
    </tr>
</%def>
```

To render a very basic template, you can do the following:

```
from mako.template import Template
print(Template("hello ${data}!").render(data="world"))
```

Mako is well respected within the Python web community.

References

HTML Scraping

Web Scraping

Web sites are written using HTML, which means that each web page is a structured document. Sometimes it would be great to obtain some data from them and preserve the structure while we're at it. Web sites don't always provide their data in comfortable formats such as `csv` or `json`.

This is where web scraping comes in. Web scraping is the practice of using a computer program to sift through a web page and gather the data that you need in a format most useful to you while at the same time preserving the structure of the data.

lxml and Requests

lxml is a pretty extensive library written for parsing XML and HTML documents very quickly, even handling messed up tags in the process. We will also be using the Requests module instead of the already built-in urllib2 module due to improvements in speed and readability. You can easily install both using `pip install lxml` and `pip install requests`.

Let's start with the imports:

```
from lxml import html
import requests
```

Next we will use `requests.get` to retrieve the web page with our data, parse it using the `html` module and save the results in `tree`:

```
page = requests.get('http://econpy.pythonanywhere.com/ex/001.html')
tree = html.fromstring(page.content)
```

(We need to use `page.content` rather than `page.text` because `html.fromstring` implicitly expects `bytes` as input.)

`tree` now contains the whole HTML file in a nice tree structure which we can go over two different ways: XPath and CSSSelect. In this example, we will focus on the former.

XPath is a way of locating information in structured documents such as HTML or XML documents. A good introduction to XPath is on W3Schools .

There are also various tools for obtaining the XPath of elements such as FireBug for Firefox or the Chrome Inspector. If you're using Chrome, you can right click an element, choose 'Inspect element', highlight the code, right click again and choose 'Copy XPath'.

After a quick analysis, we see that in our page the data is contained in two elements - one is a div with title 'buyer-name' and the other is a span with class 'item-price':

```
<div title="buyer-name">Carson Busses</div>
<span class="item-price">$29.95</span>
```

Knowing this we can create the correct XPath query and use the lxml `xpath` function like this:

```
#This will create a list of buyers:
buyers = tree.xpath('//div[@title="buyer-name"]/text()')
#This will create a list of prices
prices = tree.xpath('//span[@class="item-price"]/text()')
```

Let's see what we got exactly:

```
print 'Buyers: ', buyers
print 'Prices: ', prices
```

```
Buyers:  ['Carson Busses', 'Earl E. Byrd', 'Patty Cakes',
'Derri Anne Connecticut', 'Moe Dess', 'Leda Doggslife', 'Dan Druff',
'Al Fresco', 'Ido Hoe', 'Howie Kisses', 'Len Lease', 'Phil Meup',
'Ira Pent', 'Ben D. Rules', 'Ave Sectomy', 'Gary Shattire',
'Bobbi Soks', 'Sheila Takya', 'Rose Tattoo', 'Moe Tell']

Prices:  ['$29.95', '$8.37', '$15.26', '$19.25', '$19.25',
'$13.99', '$31.57', '$8.49', '$14.47', '$15.86', '$11.11',
'$15.98', '$16.27', '$7.50', '$50.85', '$14.26', '$5.68',
'$15.00', '$114.07', '$10.09']
```

Congratulations! We have successfully scraped all the data we wanted from a web page using lxml and Requests. We have it stored in memory as two lists. Now we can do all sorts of cool stuff with it: we can analyze it using Python or we can save it to a file and share it with the world.

Some more cool ideas to think about are modifying this script to iterate through the rest of the pages of this example dataset, or rewriting this application to use threads for improved speed.

Command-line Applications

Command-line applications, also referred to as Console Applications, are computer programs designed to be used from a text interface, such as a shell. Command-line applications usually accept various inputs as arguments, often referred to as parameters or sub-commands, as well as options, often referred to as flags or switches.

Some popular command-line applications include:

- Grep - A plain-text data search utility
- curl - A tool for data transfer with URL syntax
- httpie - A command line HTTP client, a user-friendly cURL replacement
- git - A distributed version control system
- mercurial - A distributed version control system primarily written in Python

Clint

clint is a Python module which is filled with very useful tools for developing command-line applications. It supports features such as; CLI colors and indents, simple and powerful column printer, iterator based progress bars and implicit argument handling.

Click

click is a Python package for creating command-line interfaces in a composable way with as little code as possible. This "Command-line Interface Creation Kit" is highly configurable but comes with good defaults out of the box.

docopt

docopt is a lightweight, highly Pythonic package that allows creating command-line interfaces easily and intuitively, by parsing POSIX-style usage instructions.

Plac

Plac is a simple wrapper over the Python standard library argparse, which hides most of its complexity by using a declarative interface: the argument parser is inferred rather than written down by imperatively. This module targets especially unsophisticated users, programmers, sys-admins, scientists and in general people writing throw-away scripts for themselves, who choose to create a command-line interface because it is quick and simple.

Cliff

Cliff is a framework for building command-line programs. It uses setuptools entry points to provide subcommands, output formatters, and other extensions. The framework is meant to be used to create multi-level commands such as subversion and git, where the main program handles some basic argument parsing and then invokes a sub-command to do the work.

Cement

Cement is an advanced CLI Application Framework. Its goal is to introduce a standard, and feature-full platform for both simple and complex command line applications as well as support rapid development needs without sacrificing quality. Cement is flexible, and it's use cases span from the simplicity of a micro-framework to the complexity of a meg-framework.

GUI Applications

Alphabetical list of GUI Applications.

Camelot

Camelot provides components for building applications on top of Python, SQLAlchemy and Qt. It is inspired by the Django admin interface.

The main resource for information is the website: http://www.python-camelot.com and the mailing list https://groups.google.com/forum/#!forum/project-camelot

Cocoa

Note: The Cocoa framework is only available on OS X. Don't pick this if you're writing a cross-platform application!

GTk

PyGTK provides Python bindings for the GTK+ toolkit. Like the GTK+ library itself, it is currently licensed under the GNU LGPL. It is worth noting that PyGTK only currently supports the Gtk-2.X API (NOT Gtk-3.0). It is currently recommended that PyGTK not be used for new projects and that existing applications be ported from PyGTK to PyGObject.

PyGObject aka (PyGi)

PyGObject provides Python bindings, which gives access to the entire GNOME software platform. It is fully compatible with GTK+ 3. Here is a tutorial to get started with Python GTK+ 3 Tutorial.

API Reference

Kivy

Kivy is a Python library for development of multi-touch enabled media rich applications. The aim is to allow for quick and easy interaction design and rapid prototyping, while making your code reusable and deployable.

Kivy is written in Python, based on OpenGL and supports different input devices such as: Mouse, Dual Mouse, TUIO, WiiMote, WM_TOUCH, HIDtouch, Apple's products and so on.

Kivy is actively being developed by a community and is free to use. It operates on all major platforms (Linux, OSX, Windows, Android).

The main resource for information is the website: http://kivy.org

PyObjC

Note: Only available on OS X. Don't pick this if you're writing a cross-platform application.

PySide

PySide is a Python binding of the cross-platform GUI toolkit Qt.

```
pip install pyside
```

https://wiki.qt.io/Category:LanguageBindings::PySide::Downloads

PyQt

Note: If your software does not fully comply with the GPL you will need a commercial license!

PyQt provides Python bindings for the Qt Framework (see below).

http://www.riverbankcomputing.co.uk/software/pyqt/download

PyjamasDesktop (pyjs Desktop)

PyjamasDesktop is a port of Pyjamas. PyjamasDesktop is application widget set for desktop and a cross-platform framework. (After release v0.6 PyjamasDesktop is a part of Pyjamas (Pyjs)). Briefly, it allows the exact same Python web application source code to be executed as a standalone desktop application.

Python Wiki for PyjamasDesktop.

The main website; pyjs Desktop.

Qt

Qt is a cross-platform application framework that is widely used for developing software with a GUI but can also be used for non-GUI applications.

Tk

Tkinter is a thin object-oriented layer on top of Tcl/Tk. **It has the advantage of being included with the Python standard library, making it the most convenient and compatible toolkit to program with.**

Both Tk and Tkinter are available on most Unix platforms, as well as on Windows and Macintosh systems. Starting with the 8.0 release, Tk offers native look and feel on all platforms.

There's a good multi-language Tk tutorial with Python examples at TkDocs. There's more information available on the Python Wiki.

wxPython

wxPython is a GUI toolkit for the Python programming language. It allows Python programmers to create programs with a robust, highly functional graphical user interface, simply and easily. It is implemented as a Python extension module (native code) that wraps the popular wxWidgets cross platform GUI library, which is written in C++.

Install (Stable) wxPython *go to http://www.wxpython.org/download.php#stable and download the appropriate package for your OS.*

Databases

DB-API

The Python Database API (DB-API) defines a standard interface for Python database access modules. It's documented in **PEP 249**. Nearly all Python database modules such as *sqlite3*, *psycopg* and *mysql-python* conform to this interface.

Tutorials that explain how to work with modules that conform to this interface can be found here and here.

SQLAlchemy

SQLAlchemy is a commonly used database toolkit. Unlike many database libraries it not only provides an ORM layer but also a generalized API for writing database-agnostic code without SQL.

```
$ pip install sqlalchemy
```

Records

Records is minimalist SQL library, designed for sending raw SQL queries to various databases. Data can be used programmatically, or exported to a number of useful data formats.

```
$ pip install records
```

Also included is a command-line tool for exporting SQL data.

Django ORM

The Django ORM is the interface used by Django to provide database access.

It's based on the idea of models, an abstraction that makes it easier to manipulate data in Python.

The basics:

- Each model is a Python class that subclasses django.db.models.Model.
- Each attribute of the model represents a database field.
- Django gives you an automatically-generated database-access API; see Making queries.

peewee

peewee is another ORM with a focus on being lightweight with support for Python 2.6+ and 3.2+ which supports SQLite, MySQL and Postgres by default. The model layer is similar to that of the Django ORM and it has SQL-like methods to query data. While SQLite, MySQL and Postgres are supported out-of-the-box, there is a collection of add-ons available.

PonyORM

PonyORM is an ORM that takes a different approach to querying the database. Instead of writing an SQL-like language or boolean expressions, Python's generator syntax is used. There's also an graphical schema editor that can generate PonyORM entities for you. It supports Python 2.6+ and Python 3.3+ and can connect to SQLite, MySQL, Postgres & Oracle

SQLObject

SQLObject is yet another ORM. It supports a wide variety of databases: Common database systems MySQL, Postgres and SQLite and more exotic systems like SAP DB, SyBase and MSSQL. It only supports Python 2 from Python 2.6 upwards.

Networking

Twisted

Twisted is an event-driven networking engine. It can be used to build applications around many different networking protocols, including http servers and clients, applications using SMTP, POP3, IMAP or SSH protocols, instant messaging and much more.

PyZMQ

PyZMQ is the Python binding for ZeroMQ, which is a high-performance asynchronous messaging library. One great advantage of ZeroMQ is that it can be used for message queuing without a message broker. The basic patterns for this are:

- request-reply: connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- publish-subscribe: connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- push-pull (or pipeline): connects nodes in a fan-out / fan-in pattern that can have multiple steps, and loops. This is a parallel task distribution and collection pattern.

For a quick start, read the ZeroMQ guide.

gevent

gevent is a coroutine-based Python networking library that uses greenlets to provide a high-level synchronous API on top of the libev event loop.

Systems Administration

Fabric

Fabric is a library for simplifying system administration tasks. While Chef and Puppet tend to focus on managing servers and system libraries, Fabric is more focused on application level tasks such as deployment.

Install Fabric:

```
$ pip install fabric
```

The following code will create two tasks that we can use: `memory_usage` and `deploy`. The former will output the memory usage on each machine. The latter will ssh into each server, cd to our project directory, activate the virtual environment, pull the newest codebase, and restart the application server.

```
from fabric.api import cd, env, prefix, run, task

env.hosts = ['my_server1', 'my_server2']

@task
def memory_usage():
    run('free -m')

@task
def deploy():
    with cd('/var/www/project-env/project'):
        with prefix('. ../bin/activate'):
            run('git pull')
            run('touch app.wsgi')
```

With the previous code saved in a file named `fabfile.py`, we can check memory usage with:

```
$ fab memory_usage
[my_server1] Executing task 'memory'
[my_server1] run: free -m
[my_server1] out:              total     used     free   shared  buffers   cached
[my_server1] out: Mem:          6964     1897     5067        0      166      222
[my_server1] out: -/+ buffers/cache:     1509     5455
[my_server1] out: Swap:            0        0        0

[my_server2] Executing task 'memory'
[my_server2] run: free -m
[my_server2] out:              total     used     free   shared  buffers   cached
[my_server2] out: Mem:          1666      902      764        0      180      572
[my_server2] out: -/+ buffers/cache:      148     1517
[my_server2] out: Swap:          895        1      894
```

and we can deploy with:

```
$ fab deploy
```

Additional features include parallel execution, interaction with remote programs, and host grouping.

Fabric Documentation

Salt

Salt is an open source infrastructure management tool. It supports remote command execution from a central point (master host) to multiple hosts (minions). It also supports system states which can be used to configure multiple servers using simple template files.

Salt supports Python versions 2.6 and 2.7 and can be installed via pip:

```
$ pip install salt
```

After configuring a master server and any number of minion hosts, we can run arbitrary shell commands or use pre-built modules of complex commands on our minions.

The following command lists all available minion hosts, using the ping module.

```
$ salt '*' test.ping
```

The host filtering is accomplished by matching the minion id, or using the grains system. The grains system uses static host information like the operating system version or the CPU architecture to provide a host taxonomy for the Salt modules.

The following command lists all available minions running CentOS using the grains system:

```
$ salt -G 'os:CentOS' test.ping
```

Salt also provides a state system. States can be used to configure the minion hosts.

For example, when a minion host is ordered to read the following state file, it will install and start the Apache server:

```
apache:
  pkg:
    - installed
  service:
    - running
    - enable: True
    - require:
      - pkg: apache
```

State files can be written using YAML, the Jinja2 template system or pure Python.

Salt Documentation

Psutil

Psutil is an interface to different system information (e.g. CPU, memory, disks, network, users and processes).

Here is an example to be aware of some server overload. If any of the tests (net, CPU) fail, it will send an email.

```
# Functions to get system values:
from psutil import cpu_percent, net_io_counters
# Functions to take a break:
from time import sleep
# Package for email services:
import smtplib
import string
MAX_NET_USAGE = 400000
MAX_ATTACKS = 4
```

```
attack = 0
counter = 0
while attack <= MAX_ATTACKS:
    sleep(4)
    counter = counter + 1
    # Check the cpu usage
    if cpu_percent(interval = 1) > 70:
        attack = attack + 1
    # Check the net usage
    neti1 = net_io_counters()[1]
    neto1 = net_io_counters()[0]
    sleep(1)
    neti2 = net_io_counters()[1]
    neto2 = net_io_counters()[0]
    # Calculate the bytes per second
    net = ((neti2+neto2) - (neti1+neto1))/2
    if net > MAX_NET_USAGE:
        attack = attack + 1
    if counter > 25:
        attack = 0
        counter = 0
# Write a very important email if attack is higher than 4
TO = "you@your_email.com"
FROM = "webmaster@your_domain.com"
SUBJECT = "Your domain is out of system resources!"
text = "Go and fix your server!"
BODY = string.join(("From: %s" %FROM,"To: %s" %TO,"Subject: %s" %SUBJECT, "",text), "\r\n")
server = smtplib.SMTP('127.0.0.1')
server.sendmail(FROM, [TO], BODY)
server.quit()
```

A full terminal application like a widely extended top which is based on psutil and with the ability of a client-server monitoring is glance.

Ansible

Ansible is an open source system automation tool. The biggest advantage over Puppet or Chef is it does not require an agent on the client machine. Playbooks are Ansible's configuration, deployment, and orchestration language and are written in YAML with Jinja2 for templating.

Ansible supports Python versions 2.6 and 2.7 and can be installed via pip:

```
$ pip install ansible
```

Ansible requires an inventory file that describes the hosts to which it has access. Below is an example of a host and playbook that will ping all the hosts in the inventory file.

Here is an example inventory file: `hosts.yml`

```
[server_name]
127.0.0.1
```

Here is an example playbook: `ping.yml`

```
---
- hosts: all

  tasks:
```

```
    - name: ping
      action: ping
```

To run the playbook:

```
$ ansible-playbook ping.yml -i hosts.yml --ask-pass
```

The Ansible playbook will ping all of the servers in the `hosts.yml` file. You can also select groups of servers using Ansible. For more information about Ansible, read the Ansible Docs.

An Ansible tutorial is also a great and detailed introduction to getting started with Ansible.

Chef

Chef is a systems and cloud infrastructure automation framework that makes it easy to deploy servers and applications to any physical, virtual, or cloud location. In case this is your choice for configuration management, you will primarily use Ruby to write your infrastructure code.

Chef clients run on every server that is part of your infrastructure and these regularly check with your Chef server to ensure your system is always aligned and represents the desired state. Since each individual server has its own distinct Chef client, each server configures itself and this distributed approach makes Chef a scalable automation platform.

Chef works by using custom recipes (configuration elements), implemented in cookbooks. Cookbooks, which are basically packages for infrastructure choices, are usually stored in your Chef server. Read the Digital Ocean tutorial series on chef to learn how to create a simple Chef Server.

To create a simple cookbook the knife command is used:

```
knife cookbook create cookbook_name
```

Getting started with Chef is a good starting point for Chef Beginners and many community maintained cookbooks that can serve as a good reference or tweaked to serve your infrastructure configuration needs can be found on the Chef Supermarket.

- Chef Documentation

Puppet

Puppet is IT Automation and configuration management software from Puppet Labs that allows System Administrators to define the state of their IT Infrastructure, thereby providing an elegant way to manage their fleet of physical and virtual machines.

Puppet is available both as an Open Source and an Enterprise variant. Modules are small, shareable units of code written to automate or define the state of a system. Puppet Forge is a repository for modules written by the community for Open Source and Enterprise Puppet.

Puppet Agents are installed on nodes whose state needs to be monitored or changed. A designated server known as the Puppet Master is responsible for orchestrating the agent nodes.

Agent nodes send basic facts about the system such as to the operating system, kernel, architecture, ip address, hostname etc. to the Puppet Master. The Puppet Master then compiles a catalog with information provided by the agents on how each node should be configured and sends it to the agent. The agent enforces the change as prescribed in the catalog and sends a report back to the Puppet Master.

Facter is an interesting tool that ships with Puppet that pulls basic facts about the system. These facts can be referenced as a variable while writing your Puppet modules.

```
$ facter kernel
Linux
```

```
$ facter operatingsystem
Ubuntu
```

Writing Modules in Puppet is pretty straight forward. Puppet Manifests together form Puppet Modules. Puppet manifest end with an extension of `.pp`. Here is an example of 'Hello World' in Puppet.

```
notify { 'This message is getting logged into the agent node':

    #As nothing is specified in the body the resource title
    #the notification message by default.
}
```

Here is another example with system based logic. Note how the operating system fact is being used as a variable prepended with the `$` sign. Similarly, this holds true for other facts such as hostname which can be referenced by `$hostname`

```
notify{ 'Mac Warning':
    message => $operatingsystem ? {
        'Darwin' => 'This seems to be a Mac.',
        default  => 'I am a PC.',
    },
}
```

There are several resource types for Puppet but the package-file-service paradigm is all you need for undertaking majority of the configuration management. The following Puppet code makes sure that the OpenSSH-Server package is installed in a system and the sshd service is notified to restart everytime the sshd configuration file is changed.

```
package { 'openssh-server':
    ensure => installed,
}

file { '/etc/ssh/sshd_config':
    source   => 'puppet:///modules/sshd/sshd_config',
    owner    => 'root',
    group    => 'root',
    mode     => '640',
    notify   =>  Service['sshd'], # sshd will restart
                                  # whenever you edit this
                                  # file
    require  => Package['openssh-server'],

}

service { 'sshd':
    ensure    => running,
    enable    => true,
    hasstatus => true,
    hasrestart=> true,
}
```

For more information, refer to the Puppet Labs Documentation

Blueprint

Todo

Write about Blueprint

Buildout

Buildout is an open source software build tool. Buildout is created using the Python programming language. It implements a principle of separation of configuration from the scripts that do the setting up. Buildout is primarily used to download and set up dependencies in Python eggs format of the software being developed or deployed. Recipes for build tasks in any environment can be created, and many are already available.

Shinken

Shinken is a modern, Nagios compatible monitoring framework written in Python. Its main goal is to give users a flexible architecture for their monitoring system that is designed to scale to large environments.

Shinken is backwards-compatible with the Nagios configuration standard, and plugins.It works on any operating system, and architecture that supports Python which includes Windows, GNU/Linux, and FreeBSD.

Continuous Integration

Why?

Martin Fowler, who first wrote about Continuous Integration (short: CI) together with Kent Beck, describes the CI as follows:

> Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Jenkins

Jenkins CI is an extensible continuous integration engine. Use it.

Buildbot

Buildbot is a Python system to automate the compile/test cycle to validate code changes.

Tox

tox is an automation tool providing packaging, testing and deployment of Python software right from the console or CI server. It is a generic virtualenv management and test command line tool which provides the following features:

- Checking that packages install correctly with different Python versions and interpreters
- Running tests in each of the environments, configuring your test tool of choice

- Acting as a front-end to Continuous Integration servers, reducing boilerplate and merging CI and shell-based testing.

Travis-CI

Travis-CI is a distributed CI server which builds tests for open source projects for free. It provides multiple workers to run Python tests on and seamlessly integrates with GitHub. You can even have it comment on your Pull Requests whether this particular changeset breaks the build or not. So if you are hosting your code on GitHub, travis-ci is a great and easy way to get started with Continuous Integration.

In order to get started, add a `.travis.yml` file to your repository with this example content:

```
language: python
python:
  - "2.6"
  - "2.7"
  - "3.2"
  - "3.3"
# command to install dependencies
script: python tests/test_all_of_the_units.py
branches:
  only:
    - master
```

This will get your project tested on all the listed Python versions by running the given script, and will only build the master branch. There are a lot more options you can enable, like notifications, before and after steps and much more. The travis-ci docs explain all of these options, and are very thorough.

In order to activate testing for your project, go to the travis-ci site and login with your GitHub account. Then activate your project in your profile settings and you're ready to go. From now on, your project's tests will be run on every push to GitHub.

Speed

CPython, the most commonly used implementation of Python, is slow for CPU bound tasks. PyPy is fast.

Using a slightly modified version of David Beazley's CPU bound test code (added loop for multiple tests), you can see the difference between CPython and PyPy's processing.

```
# PyPy
$ ./pypy -V
Python 2.7.1 (7773f8fc4223, Nov 18 2011, 18:47:10)
[PyPy 1.7.0 with GCC 4.4.3]
$ ./pypy measure2.py
0.0683999061584
0.0483210086823
0.0388588905334
0.0440690517426
0.0695300102234
```

```
# CPython
$ ./python -V
Python 2.7.1
$ ./python measure2.py
1.06774401665
1.45412397385
```

```
1.51485204697
1.54693889618
1.60109114647
```

Context

The GIL

The GIL (Global Interpreter Lock) is how Python allows multiple threads to operate at the same time. Python's memory management isn't entirely thread-safe, so the GIL is required to prevent multiple threads from running the same Python code at once.

David Beazley has a great guide on how the GIL operates. He also covers the new GIL in Python 3.2. His results show that maximizing performance in a Python application requires a strong understanding of the GIL, how it affects your specific application, how many cores you have, and where your application bottlenecks are.

C Extensions

The GIL

Special care must be taken when writing C extensions to make sure you register your threads with the interpreter.

C Extensions

Cython

Cython implements a superset of the Python language with which you are able to write C and C++ modules for Python. Cython also allows you to call functions from compiled C libraries. Using Cython allows you to take advantage of Python's strong typing of variables and operations.

Here's an example of strong typing with Cython:

```
def primes(int kmax):
"""Calculation of prime numbers with additional
Cython keywords"""

    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

This implementation of an algorithm to find prime numbers has some additional keywords compared to the next one, which is implemented in pure Python:

```
def primes(kmax):
"""Calculation of prime numbers in standard Python syntax"""

    p = range(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Notice that in the Cython version you declare integers and integer arrays to be compiled into C types while also creating a Python list:

```
def primes(int kmax):
    """Calculation of prime numbers with additional
    Cython keywords"""

    cdef int n, k, i
    cdef int p[1000]
    result = []
```

```
def primes(kmax):
    """Calculation of prime numbers in standard Python syntax"""

    p = range(1000)
    result = []
```

What is the difference? In the upper Cython version you can see the declaration of the variable types and the integer array in a similar way as in standard C. For example *cdef int n,k,i* in line 3. This additional type declaration (i.e. integer) allows the Cython compiler to generate more efficient C code from the second version. While standard Python code is saved in `*.py` files, Cython code is saved in `*.pyx` files.

What's the difference in speed? Let's try it!

```
import time
#activate pyx compiler
import pyximport
pyximport.install()
#primes implemented with Cython
import primesCy
#primes implemented with Python
import primes

print "Cython:"
t1= time.time()
print primesCy.primes(500)
t2= time.time()
```

```
print "Cython time: %s" %(t2-t1)
print ""
print "Python"
t1= time.time()
print primes.primes(500)
t2= time.time()
print "Python time: %s" %(t2-t1)
```

These lines both need a remark:

```
import pyximport
pyximport.install()
```

The *pyximport* module allows you to import `*.pyx` files (e.g., `primesCy.pyx`) with the Cython-compiled version of the *primes* function. The *pyximport.install()* command allows the Python interpreter to start the Cython compiler directly to generate C-code, which is automatically compiled to a `*.so` C-library. Cython is then able to import this library for you in your Python code, easily and efficiently. With the *time.time()* function you are able to compare the time between these 2 different calls to find 500 prime numbers. On a standard notebook (dual core AMD E-450 1.6 GHz), the measured values are:

```
Cython time: 0.0054 seconds

Python time: 0.0566 seconds
```

And here the output of an embedded ARM beaglebone machine:

```
Cython time: 0.0196 seconds

Python time: 0.3302 seconds
```

Pyrex

Shedskin?

Concurrency

Concurrent.futures

The concurrent.futures module is a module in the standard library that provides a "high-level interface for asynchronously executing callables". It abstracts away a lot of the more complicated details about using multiple threads or processes for concurrency, and allows the user to focus on accomplishing the task at hand.

The concurrent.futures module exposes two main classes, the *ThreadPoolExecutor* and the *ProcessPoolExecutor*. The ThreadPoolExecutor will create a pool of worker threads that a user can submit jobs to. These jobs will then be executed in another thread when the next worker thread becomes available.

The ProcessPoolExecutor works in the same way, except instead of using multiple threads for its workers, it will use multiple processes. This makes it possible to side-step the GIL, however because of the way things are passed to worker processes, only picklable objects can be executed and returned.

Because of the way the GIL works, a good rule of thumb is to use a ThreadPoolExecutor when the task being executed involves a lot of blocking (i.e. making requests over the network) and to use a ProcessPoolExecutor executor when the task is computationally expensive.

There are two main ways of executing things in parallel using the two Executors. One way is with the *map(func, iterables)* method. This works almost exactly like the builtin *map()* function, except it will execute everything in parallel. :

```
from concurrent.futures import ThreadPoolExecutor
import requests

def get_webpage(url):
    page = requests.get(url)
    return page

pool = ThreadPoolExecutor(max_workers=5)

my_urls = ['http://google.com/']*10  # Create a list of urls

for page in pool.map(get_webpage, my_urls):
    # Do something with the result
    print(page.text)
```

For even more control, the *submit(func, *args, **kwargs)* method will schedule a callable to be executed (as *func(*args, **kwargs)*) and returns a Future object that represents the execution of the callable.

The Future object provides various methods that can be used to check on the progress of the scheduled callable. These include:

cancel() Attempt to cancel the call.

cancelled() Return True if the call was successfully cancelled.

running() Return True if the call is currently being executed and cannot be cancelled.

done() Return True if the call was successfully cancelled or finished running.

result() Return the value returned by the call. Note that this call will block until the scheduled callable returns by default.

exception() Return the exception raised by the call. If no exception was raised then this returns *None*. Note that this will block just like *result()*.

add_done_callback(fn) Attach a callback function that will be executed (as *fn(future)*) when the scheduled callable returns.

```
from concurrent.futures import ProcessPoolExecutor, as_completed

def is_prime(n):
    if n % 2 == 0:
        return n, False

    sqrt_n = int(n**0.5)
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return n, False
    return n, True

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

futures = []
with ProcessPoolExecutor(max_workers=4) as pool:
    # Schedule the ProcessPoolExecutor to check if a number is prime
```

```
    # and add the returned Future to our list of futures
    for p in PRIMES:
        fut = pool.submit(is_prime, p)
        futures.append(fut)

# As the jobs are completed, print out the results
for number, result in as_completed(futures):
    if result:
        print("{} is prime".format(number))
    else:
        print("{} is not prime".format(number))
```

The concurrent.futures module contains two helper functions for working with Futures. The *as_completed(futures)* function returns an iterator over the list of futures, yielding the futures as they complete.

The *wait(futures)* function will simply block until all futures in the list of futures provided have completed.

For more information, on using the concurrent.futures module, consult the official documentation.

Threading

The standard library comes with a threading module that allows a user to work with multiple threads manually.

Running a function in another thread is as simple as passing a callable and it's arguments to *Thread*'s constructor and then calling *start()*:

```
from threading import Thread
import requests

def get_webpage(url):
    page = requests.get(url)
    return page

some_thread = Thread(get_webpage, 'http://google.com/')
some_thread.start()
```

To wait until the thread has terminated, call *join()*:

```
some_thread.join()
```

After calling *join()*, it is always a good idea to check whether the thread is still alive (because the join call timed out):

```
if some_thread.is_alive():
    print("join() must have timed out.")
else:
    print("Our thread has terminated.")
```

Because multiple threads have access to the same section of memory, sometimes there might be situations where two or more threads are trying to write to the same resource at the same time or where the output is dependent on the sequence or timing of certain events. This is called a data race or race condition. When this happens, the output will be garbled or you may encounter problems which are difficult to debug. A good example is this stackoverflow post.

The way this can be avoided is by using a **'Lock'_** that each thread needs to acquire before writing to a shared resource. Locks can be acquired and released through either the contextmanager protocol (*with* statement), or by using *acquire()* and *release()* directly. Here is a (rather contrived) example:

```
from threading import Lock, Thread

file_lock = Lock()
```

```
def log(msg):
    with file_lock:
        open('website_changes.log', 'w') as f:
            f.write(changes)

def monitor_website(some_website):
    """
    Monitor a website and then if there are any changes,
    log them to disk.
    """
    while True:
        changes = check_for_changes(some_website)
        if changes:
            log(changes)

websites = ['http://google.com/', ... ]
for website in websites:
    t = Thread(monitor_website, website)
    t.start()
```

Here, we have a bunch of threads checking for changes on a list of sites and whenever there are any changes, they attempt to write those changes to a file by calling *log(changes)*. When *log()* is called, it will wait to acquire the lock with *with file_lock:*. This ensures that at any one time, only one thread is writing to the file.

Spawning Processes

Multiprocessing

Scientific Applications

Context

Python is frequently used for high-performance scientific applications. It is widely used in academia and scientific projects because it is easy to write and performs well.

Due to its high performance nature, scientific computing in Python often utilizes external libraries, typically written in faster languages (like C, or FORTRAN for matrix operations). The main libraries used are NumPy, SciPy and Matplotlib. Going into detail about these libraries is beyond the scope of the Python guide. However, a comprehensive introduction to the scientific Python ecosystem can be found in the Python Scientific Lecture Notes

Tools

IPython

IPython is an enhanced version of Python interpreter, which provides features of great interest to scientists. The *inline mode* allows graphics and plots to be displayed in the terminal (Qt based version). Moreover, the *notebook* mode supports literate programming and reproducible science generating a web-based Python notebook. This notebook allows you to store chunks of Python code along side the results and additional comments (HTML, LaTeX, Markdown). The notebook can then be shared and exported in various file formats.

Libraries

NumPy

NumPy is a low level library written in C (and FORTRAN) for high level mathematical functions. NumPy cleverly overcomes the problem of running slower algorithms on Python by using multidimensional arrays and functions that operate on arrays. Any algorithm can then be expressed as a function on arrays, allowing the algorithms to be run quickly.

NumPy is part of the SciPy project, and is released as a separate library so people who only need the basic requirements can use it without installing the rest of SciPy.

NumPy is compatible with Python versions 2.4 through to 2.7.2 and 3.1+.

Numba

Numba is a NumPy aware Python compiler (just-in-time (JIT) specializing compiler) which compiles annotated Python (and NumPy) code to LLVM (Low Level Virtual Machine) through special decorators. Briefly, Numba uses a system that compiles Python code with LLVM to code which can be natively executed at runtime.

SciPy

SciPy is a library that uses NumPy for more mathematical functions. SciPy uses NumPy arrays as the basic data structure, and comes with modules for various commonly used tasks in scientific programming, including linear algebra, integration (calculus), ordinary differential equation solving and signal processing.

Matplotlib

Matplotlib is a flexible plotting library for creating interactive 2D and 3D plots that can also be saved as manuscript-quality figures. The API in many ways reflects that of MATLAB, easing transition of MATLAB users to Python. Many examples, along with the source code to re-create them, are available in the matplotlib gallery.

Pandas

Pandas is data manipulation library based on Numpy which provides many useful functions for accessing, indexing, merging and grouping data easily. The main data structure (DataFrame) is close to what could be found in the R statistical package; that is, heterogeneous data tables with name indexing, time series operations and auto-alignment of data.

Rpy2

Rpy2 is a Python binding for the R statistical package allowing the execution of R functions from Python and passing data back and forth between the two environments. Rpy2 is the object oriented implementation of the Rpy bindings.

PsychoPy

PsychoPy is a library for cognitive scientists allowing the creation of cognitive psychology and neuroscience experiments. The library handles presentation of stimuli, scripting of experimental design and data collection.

Resources

Installation of scientific Python packages can be troublesome, as many of these packages are implemented as Python C extensions which need to be compiled. This section lists various so-called scientific Python distributions which provide precompiled and easy-to-install collections of scientific Python packages.

Unofficial Windows Binaries for Python Extension Packages

Many people who do scientific computing are on Windows, yet many of the scientific computing packages are notoriously difficult to build and install on this platform. Christoph Gohlke however, has compiled a list of Windows binaries for many useful Python packages. The list of packages has grown from a mainly scientific Python resource to a more general list. If you're on Windows, you may want to check it out.

Anaconda

Continuum Analytics offers the Anaconda Python Distribution which includes all the common scientific Python packages as well as many packages related to data analytics and big data. Anaconda itself is free, and Continuum sells a number of proprietary add-ons. Free licenses for the add-ons are available for academics and researchers.

Canopy

Canopy is another scientific Python distribution, produced by Enthought. A limited 'Canopy Express' variant is available for free, but Enthought charges for the full distribution. Free licenses are available for academics.

Image Manipulation

Most image processing and manipulation techniques can be carried out effectively using two libraries: Python Imaging Library (PIL) and OpenSource Computer Vision (OpenCV).

A brief description of both is given below.

Python Imaging Library

The Python Imaging Library, or PIL for short, is one of the core libraries for image manipulation in Python. Unfortunately, its development has stagnated, with its last release in 2009.

Luckily for you, there's an actively-developed fork of PIL called Pillow - it's easier to install, runs on all operating systems, and supports Python 3.

Installation

Before installing Pillow, you'll have to install Pillow's prerequisites. Find the instructions for your platform in the Pillow installation instructions.

After that, it's straightforward:

```
$ pip install Pillow
```

Example

```
from PIL import Image, ImageFilter
#Read image
im = Image.open( 'image.jpg' )
#Display image
im.show()

#Applying a filter to the image
im_sharp = im.filter( ImageFilter.SHARPEN )
#Saving the filtered image to a new file
im_sharp.save( 'image_sharpened.jpg', 'JPEG' )

#Splitting the image into its respective bands, i.e. Red, Green,
#and Blue for RGB
r,g,b = im_sharp.split()

#Viewing EXIF data embedded in image
exif_data = im._getexif()
exif_data
```

There are more examples of the Pillow library in the Pillow tutorial.

OpenSource Computer Vision

OpenSource Computer Vision, more commonly known as OpenCV, is a more advanced image manipulation and processing software than PIL. It has been implemented in several languages and is widely used.

Installation

In Python, image processing using OpenCV is implemented using the `cv2` and `NumPy` modules. The installation instructions for OpenCV should guide you through configuring the project for yourself.

NumPy can be downloaded from the Python Package Index(PyPI):

```
$ pip install numpy
```

Example

```
from cv2 import *
import numpy as np
#Read Image
img = cv2.imread('testimg.jpg')
#Display Image
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()

#Applying Grayscale filter to image
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

#Saving filtered image to new file
cv2.imwrite('graytest.jpg',gray)
```

There are more Python-implemented examples of OpenCV in this collection of tutorials.

Data Serialization

What is data serialization?

Data serialization is the concept of converting structured data into a format that allows it to be shared or stored in such a way that its original structure to be recovered. In some cases, the secondary intention of data serialization is to minimize the size of the serialized data which then minimizes disk space or bandwidth requirements.

Pickle

The native data serialization module for Python is called Pickle.

Here's an example:

```
import pickle

#Here's an example dict
grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }

#Use dumps to convert the object to a serialized string
serial_grades = pickle.dumps( grades )

#Use loads to de-serialize an object
received_grades = pickle.loads( serial_grades )
```

Protobuf

If you're looking for a serialization module that has support in multiple languages, Google's Protobuf library is an option.

XML parsing

untangle

untangle is a simple library which takes an XML document and returns a Python object which mirrors the nodes and attributes in its structure.

For example, an XML file like this:

```
<?xml version="1.0"?>
<root>
    <child name="child1">
</root>
```

can be loaded like this:

```
import untangle
obj = untangle.parse('path/to/file.xml')
```

and then you can get the child elements name like this:

```
obj.root.child['name']
```

untangle also supports loading XML from a string or an URL.

xmltodict

xmltodict is another simple library that aims at making XML feel like working with JSON.

An XML file like this:

```
<mydocument has="an attribute">
  <and>
    <many>elements</many>
    <many>more elements</many>
  </and>
  <plus a="complex">
    element as well
  </plus>
</mydocument>
```

can be loaded into a Python dict like this:

```
import xmltodict

with open('path/to/file.xml') as fd:
    doc = xmltodict.parse(fd.read())
```

and then you can access elements, attributes and values like this:

```
doc['mydocument']['@has'] # == u'an attribute'
doc['mydocument']['and']['many'] # == [u'elements', u'more elements']
doc['mydocument']['plus']['@a'] # == u'complex'
doc['mydocument']['plus']['#text'] # == u'element as well'
```

xmltodict also lets you roundtrip back to XML with the unparse function, has a streaming mode suitable for handling files that don't fit in memory and supports namespaces.

JSON

The json library can parse JSON from strings or files. The library parses JSON into a Python dictionary or list. It can also convert Python dictionaries or lists into JSON strings.

Parsing JSON

Take the following string containing JSON data:

```
json_string = '{"first_name": "Guido", "last_name":"Rossum"}'
```

It can be parsed like this:

```
import json
parsed_json = json.loads(json_string)
```

and can now be used as a normal dictionary:

```
print(parsed_json['first_name'])
"Guido"
```

You can also convert the following to JSON:

```
d = {
    'first_name': 'Guido',
    'second_name': 'Rossum',
    'titles': ['BDFL', 'Developer'],
}

print(json.dumps(d))
'{"first_name": "Guido", "last_name": "Rossum", "titles": ["BDFL", "Developer"]}'
```

simplejson

The JSON library was added to Python in version 2.6. If you're using an earlier version of Python, the simplejson library is available via PyPI.

simplejson mimics the json standard library. It is available so that developers that use older versions of Python can use the latest features available in the json lib.

You can start using simplejson when the json library is not available by importing simplejson under a different name:

```
import simplejson as json
```

After importing simplejson as json, the above examples will all work as if you were using the standard json library.

Cryptography

Cryptography

Cryptography is an actively developed library that provides cryptographic recipes and primitives. It supports Python 2.6-2.7, Python 3.3+ and PyPy.

Cryptography is divided into two layers of recipes and hazardous materials (hazmat). The recipes layer provides simple API for proper symmetric encryption and the hazmat layer provides low-level cryptographic primitives.

Installation

```
$ pip install cryptography
```

Example

Example code using high level symmetric encryption recipe:

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"A really secret message. Not for prying eyes.")
plain_text = cipher_suite.decrypt(cipher_text)
```

PyCrypto

PyCrypto is another library, which provides secure hash functions and various encryption algorithms. It supports Python version 2.1 through 3.3.

Installation

```
$ pip install pycrypto
```

Example

```
from Crypto.Cipher import AES
# Encryption
encryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
cipher_text = encryption_suite.encrypt("A really secret message. Not for prying eyes.")

# Decryption
decryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
plain_text = decryption_suite.decrypt(cipher_text)
```

Machine Learning

Python has a vast number of libraries for data analysis, statistics and Machine Learning itself, making it a language of choice for many data scientists.

Some widely used packages for Machine Learning and other Data Science applications are enlisted below.

Scipy Stack

The Scipy stack consists of a bunch of core helper packages used in data science, for statistical analysis and visualising data. Because of its huge number of functionalities and ease of use, the Stack is considered a must-have for most data science applications.

The Stack consists of the following packages (link to documentation given):

1. NumPy
2. SciPy library
3. Matplotlib
4. IPython
5. pandas
6. Sympy
7. nose

The stack also comes with Python bundled in, but has been excluded from the above list.

Installation

For installing the full stack, or individual packages, you can refer to the instructions given here.

NB: Anaconda is highly preferred and recommended for installing and maintaining data science packages seamlessly.

scikit-learn

Scikit is a free and open-source machine learning library for Python. It offers off-the-shelf functions to implement many algorithms like linear regression, classifiers, SVMs, k-means, Neural Networks etc. It also has a few sample datasets which can be directly used for training and testing.

Because of its speed, robustness and easiness to use, it's one of the most widely-used libraries for many Machine Learning applications.

Installation

Through PyPI:

```
pip install -U scikit-learn
```

Through conda:

```
conda install scikit-learn
```

scikit-learn also comes in shipped with Anaconda (mentioned above). For more installation instructions, refer to this link.

Example

For this example, we train a simple classifier on the Iris dataset, which comes bundled in with scikit-learn.

The dataset takes four features of flowers: sepal length, sepal width, petal length and petal width, and classifies them into three flower species (labels): setosa, versicolor or virginica. The labels have been represented as numbers in the dataset: 0 (setosa), 1 (versicolor) and 2 (virginica).

We shuffle the Iris dataset, and divide it into separate training and testing sets: keeping the last 10 data points for testing and rest for training. We then train the classifier on the training set, and predict on the testing set.

```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.metrics import accuracy_score
import numpy as np

#loading the iris dataset
iris = load_iris()

x = iris.data #array of the data
y = iris.target #array of labels (i.e answers) of each data entry

#getting label names i.e the three flower species
y_names = iris.target_names

#taking random indices to split the dataset into train and test
test_ids = np.random.permutation(len(x))

#splitting data and labels into train and test
#keeping last 10 entries for testing, rest for training

x_train = x[test_ids[:-10]]
x_test = x[test_ids[-10:]]

y_train = y[test_ids[:-10]]
```

```
y_test = y[test_ids[-10:]]

#classifying using decision tree
clf = tree.DecisionTreeClassifier()

#training (fitting) the classifier with the training set
clf.fit(x_train, y_train)

#predictions on the test dataset
pred = clf.predict(x_test)

print pred #predicted labels i.e flower species
print y_test #actual labels
print (accuracy_score(pred, y_test))*100 #prediction accuracy
```

Since we're splitting randomly and the classifier trains on every iteration, the accuracy may vary. Running the above code gives:

```
[0 1 1 1 0 2 0 2 2 2]
[0 1 1 1 0 2 0 2 2 2]
100.0
```

The first line contains the labels (i.e flower species) of the testing data as predicted by our classifier, and the second line contains the actual flower species as given in the dataset. We thus get an accuracy of 100% this time.

More on scikit-learn can be read in the documentation.

Interfacing with C/C++ Libraries

C Foreign Function Interface

CFFI provides a simple to use mechanism for interfacing with C from both CPython and PyPy. It supports two modes: an inline ABI compatibility mode (example provided below), which allows you to dynamically load and run functions from executable modules (essentially exposing the same functionality as LoadLibrary or dlopen), and an API mode, which allows you to build C extension modules.

ABI Interaction

```
from cffi import FFI
ffi = FFI()
ffi.cdef("size_t strlen(const char*);")
clib = ffi.dlopen(None)
length = clib.strlen("String to be evaluated.")
# prints: 23
print("{}".format(length))
```

ctypes

ctypes is the de facto library for interfacing with C/C++ from CPython, and it provides not only full access to the native C interface of most major operating systems (e.g., kernel32 on Windows, or libc on *nix), but also provides support for loading and interfacing with dynamic libraries, such as DLLs or shared objects at runtime. It does bring along with it a whole host of types for interacting with system APIs, and allows you to rather easily define your own complex types, such as structs and unions, and allows you to modify things such as padding and alignment, if needed.

It can be a bit crufty to use, but in conjunction with the struct module, you are essentially provided full control over how your data types get translated into something usable by a pure C(++) method.

Struct Equivalents

MyStruct.h

```
struct my_struct {
    int a;
    int b;
};
```

MyStruct.py

```
import ctypes
class my_struct(ctypes.Structure):
    _fields_ = [("a", c_int),
                ("b", c_int)]
```

SWIG

SWIG, though not strictly Python focused (it supports a large number of scripting languages), is a tool for generating bindings for interpreted languages from C/C++ header files. It is extremely simple to use: the consumer simply needs to define an interface file (detailed in the tutorial and documentations), include the requisite C/C++ headers, and run the build tool against them. While it does have some limits, (it currently seems to have issues with a small subset of newer C++ features, and getting template-heavy code to work can be a bit verbose), it provides a great deal of power and exposes lots of features to Python with little effort. Additionally, you can easily extend the bindings SWIG creates (in the interface file) to overload operators and built-in methods, effectively re- cast C++ exceptions to be catchable by Python, etc.

Example: Overloading __repr__

MyClass.h

```
#include <string>
class MyClass {
private:
    std::string name;
public:
    std::string getName();
};
```

myclass.i

```
%include "string.i"

%module myclass
%{
#include <string>
#include "MyClass.h"
%}

%extend MyClass {
    std::string __repr__()
    {
```

```
        return $self->getName();
    }
}

%include "MyClass.h"
```

Boost.Python

Boost.Python requires a bit more manual work to expose C++ object functionality, but it is capable of providing all the same features SWIG does and then some, to include providing wrappers to access PyObjects in C++, extracting SWIG- wrapper objects, and even embedding bits of Python into your C++ code.

CHAPTER 4

Shipping Great Python Code

This part of the guide focuses on deploying your Python code.

Packaging Your Code

Package your code to share it with other developers. For example to share a library for other developers to use in their application, or for development tools like 'py.test'.

An advantage of this method of distribution is its well established ecosystem of tools such as PyPI and pip, which make it easy for other developers to download and install your package either for casual experiments, or as part of large, professional systems.

It is a well-established convention for Python code to be shared this way. If your code isn't packaged on PyPI, then it will be harder for other developers to find it, and to use it as part of their existing process. They will regard such projects with substantial suspicion of being either badly managed or abandoned.

The downside of distributing code like this is that it relies on the recipient understanding how to install the required version of Python, and being able and willing to use tools such as pip to install your code's other dependencies. This is fine when distributing to other developers, but makes this method unsuitable for distributing applications to end-users.

The Python Packaging Guide provides an extensive guide on creating and maintaining Python packages.

Alternatives to Packaging

To distribute applications to end-users, you should *freeze your application*.

On Linux, you may also want to consider *creating a Linux distro package* (e.g. a .deb file for Debian or Ubuntu.)

For Python Developers

If you're writing an open source Python module, PyPI , more properly known as *The Cheeseshop*, is the place to host it.

Pip vs. easy_install

Use pip. More details here

Personal PyPI

If you want to install packages from a source other than PyPI, (say, if your packages are *proprietary*), you can do it by hosting a simple http server, running from the directory which holds those packages which need to be installed.

Showing an example is always beneficial

For example, if you want to install a package called `MyPackage.tar.gz`, and assuming this is your directory structure:

- **archive**
  - **MyPackage**
    - MyPackage.tar.gz

Go to your command prompt and type:

```
$ cd archive
$ python -m SimpleHTTPServer 9000
```

This runs a simple http server running on port 9000 and will list all packages (like **MyPackage**). Now you can install **MyPackage** using any Python package installer. Using Pip, you would do it like:

```
$ pip install --extra-index-url=http://127.0.0.1:9000/ MyPackage
```

Having a folder with the same name as the package name is **crucial** here. I got fooled by that, one time. But if you feel that creating a folder called `MyPackage` and keeping `MyPackage.tar.gz` inside that, is *redundant*, you can still install MyPackage using:

```
$ pip install  http://127.0.0.1:9000/MyPackage.tar.gz
```

pypiserver

Pypiserver is a minimal PyPI compatible server. It can be used to serve a set of packages to easy_install or pip. It includes helpful features like an administrative command (`-U`) which will update all its packages to their latest versions found on PyPI.

S3-Hosted PyPi

One simple option for a personal PyPi server is to use Amazon S3. A prerequisite for this is that you have an Amazon AWS account with an S3 bucket.

1. **Install all your requirements from PyPi or another source**
2. **Install pip2pi**
   - `pip install git+https://github.com/wolever/pip2pi.git`
3. **Follow pip2pi README for pip2tgz and dir2pi commands**
   - `pip2tgz packages/ YourPackage` (or `pip2tgz packages/ -r requirements.txt`)
   - `dir2pi packages/`
4. **Upload the new files**
   - Use a client like Cyberduck to sync the entire `packages` folder to your s3 bucket
   - Make sure you upload `packages/simple/index.html` as well as all new files and directories

5. **Fix new file permissions**

- By default, when you upload new files to the S3 bucket, they will have the wrong permissions set.
- Use the Amazon web console to set the READ permission of the files to EVERYONE.
- If you get HTTP 403 when trying to install a package, make sure you've set the permissions correctly.

6. **All done**

- You can now install your package with `pip install --index-url=http://your-s3-bucket/packages/simple YourPackage`

For Linux Distributions

Creating a Linux distro package is arguably the "right way" to distribute code on Linux.

Because a distribution package doesn't include the Python interpreter, it makes the download and install about 2MB smaller than *freezing your application*.

Also, if a distribution releases a new security update for Python, then your application will automatically start using that new version of Python.

The bdist_rpm command makes producing an RPM file for use by distributions like Red Hat or SuSE is trivially easy.

However, creating and maintaining the different configurations required for each distribution's format (e.g. .deb for Debian/Ubuntu, .rpm for Red Hat/Fedora, etc) is a fair amount of work. If your code is an application that you plan to distribute on other platforms, then you'll also have to create and maintain the separate config required to freeze your application for Windows and OSX. It would be much less work to simply create and maintain a single config for one of the cross platform *freezing tools*, which will produce stand-alone executables for all distributions of Linux, as well as Windows and OSX.

Creating a distribution package is also problematic if your code is for a version of Python that isn't currently supported by a distribution. Having to tell *some versions* of Ubuntu end-users that they need to add the 'dead-snakes' PPA using *sudo apt-repository* commands before they can install your .deb file makes for an extremely hostile user experience. Not only that, but you'd have to maintain a custom equivalent of these instructions for every distribution, and worse, have your users read, understand, and act on them.

Having said all that, here's how to do it:

- Fedora
- Debian and Ubuntu
- Arch

Useful Tools

- fpm
- alien
- dh-virtualenv (for APT/DEB omnibus packaging)

Freezing Your Code

"Freezing" your code is creating a single-file executable file to distribute to end-users, that contains all of your application code as well as the Python interpreter.

Applications such as 'Dropbox', 'Eve Online', 'Civilization IV', and BitTorrent clients do this.

The advantage of distributing this way is that your application will "just work", even if the user doesn't already have the required version of Python (or any) installed. On Windows, and even on many Linux distributions and OS X, the right version of Python will not already be installed.

Besides, end-user software should always be in an executable format. Files ending in `.py` are for software engineers and system administrators.

One disadvantage of freezing is that it will increase the size of your distribution by about 2–12MB. Also, you will be responsible for shipping updated versions of your application when security vulnerabilities to Python are patched.

Alternatives to Freezing

Packaging your code is for distributing libraries or tools to other developers.

On Linux, an alternative to freezing is to *create a Linux distro package* (e.g. .deb files for Debian or Ubuntu, or .rpm files for Red Hat and SuSE.)

Todo

Fill in "Freezing Your Code" stub

Comparison of Freezing Tools

Solutions and platforms/features supported:

Solution	Windows	Linux	OS X	Python 3	License	One-file mode	Zipfile import	Eggs	pkg_resources support
bbFreeze	yes	yes	yes	no	MIT	no	yes	yes	yes
py2exe	yes	no	no	yes	MIT	yes	yes	no	no
pyInstaller	yes	yes	yes	yes	GPL	yes	no	yes	no
cx_Freeze	yes	yes	yes	yes	PSF	no	yes	yes	no
py2app	no	no	yes	yes	MIT	no	yes	yes	yes

Note: Freezing Python code on Linux into a Windows executable was only once supported in PyInstaller and later dropped..

Note: All solutions need MS Visual C++ dll to be installed on target machine, except py2app. Only Pyinstaller makes self-executable exe that bundles the dll when passing `--onefile` to `Configure.py`.

Windows

bbFreeze

Prerequisite is to install *Python, Setuptools and pywin32 dependency on Windows*.

Todo

Write steps for most basic .exe

py2exe

Prerequisite is to install *Python on Windows*.

1. Download and install http://sourceforge.net/projects/py2exe/files/py2exe/
2. Write `setup.py` (List of configuration options):

```
from distutils.core import setup
import py2exe

setup(
    windows=[{'script': 'foobar.py'}],
)
```

3. (Optionally) include icon
4. (Optionally) one-file mode
5. Generate `.exe` into `dist` directory:

```
$ python setup.py py2exe
```

6. Provide the Microsoft Visual C runtime DLL. Two options: globally install dll on target machine or distribute dll alongside with .exe.

PyInstaller

Prerequisite is to have installed *Python, Setuptools and pywin32 dependency on Windows*.

- Most basic tutorial
- Manual

OS X

py2app

PyInstaller

PyInstaller can be used to build Unix executables and windowed apps on Mac OS X 10.6 (Snow Leopard) or newer.

To install PyInstaller, use pip:

```
$ pip install pyinstaller
```

To create a standard Unix executable, from say `script.py`, use:

```
$ pyinstaller script.py
```

This creates,

- a `script.spec` file, analogous to a `make` file
- a `build` folder, that holds some log files

- a `dist` folder, that holds the main executable `script`, and some dependent Python libraries,

all in the same folder as `script.py`. PyInstaller puts all the Python libraries used in `script.py` into the `dist` folder, so when distributing the executable, distribute the whole `dist` folder.

The `script.spec` file can be edited to customise the build, with options such as

- bundling data files with the executable
- including run-time libraries (`.dll` or `.so` files) that PyInstaller can't infer automatically
- adding Python run-time options to the executable,

Now `script.spec` can be run with `pyinstaller` (instead of using `script.py` again):

```
$ pyinstaller script.spec
```

To create a standalone windowed OS X application, use the `--windowed` option

```
$ pyinstaller --windowed script.spec
```

This creates a `script.app` in the `dist` folder. Make sure to use GUI packages in your Python code, like PyQt or PySide, to control the graphical parts of the app.

There are several options in `script.spec` related to Mac OS X app bundles here. For example, to specify an icon for the app, use the `icon=\path\to\icon.icns` option.

Linux

bbFreeze

PyInstaller

CHAPTER 5

Python Development Environments

This part of the guide focus on the Python development environment, and the best-practice tools that are available for writing Python code.

Your Development Environment

Text Editors

Just about anything that can edit plain text will work for writing Python code, however, using a more powerful editor may make your life a bit easier.

Vim

Vim is a text editor which uses keyboard shortcuts for editing instead of menus or icons. There are a couple of plugins and settings for the Vim editor to aid Python development. If you only develop in Python, a good start is to set the default settings for indentation and line-wrapping to values compliant with **PEP 8**. In your home directory, open a file called `.vimrc` and add the following lines:

```
set textwidth=79  " lines longer than 79 columns will be broken
set shiftwidth=4  " operation >> indents 4 columns; << unindents 4 columns
set tabstop=4     " a hard TAB displays as 4 columns
set expandtab     " insert spaces when hitting TABs
set softtabstop=4 " insert/delete 4 spaces when hitting a TAB/BACKSPACE
set shiftround    " round indent to multiple of 'shiftwidth'
set autoindent    " align the new line indent with the previous line
```

With these settings, newlines are inserted after 79 characters and indentation is set to 4 spaces per tab. If you also use Vim for other languages, there is a handy plugin called indent, which handles indentation settings for Python source files.

There is also a handy syntax plugin called syntax featuring some improvements over the syntax file included in Vim 6.1.

These plugins supply you with a basic environment for developing in Python. To get the most out of Vim, you should continually check your code for syntax errors and PEP8 compliance. Luckily PEP8 and Pyflakes will do this for you. If your Vim is compiled with `+python` you can also utilize some very handy plugins to do these checks from within the editor.

For PEP8 checking and pyflakes, you can install vim-flake8. Now you can map the function `Flake8` to any hotkey or action you want in Vim. The plugin will display errors at the bottom of the screen, and provide an easy way to jump

to the corresponding line. It's very handy to call this function whenever you save a file. In order to do this, add the following line to your `.vimrc`:

```
autocmd BufWritePost *.py call Flake8()
```

If you are already using syntastic, you can set it to run Pyflakes on write and show errors and warnings in the quickfix window. An example configuration to do that which also shows status and warning messages in the statusbar would be:

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

Python-mode

Python-mode is a complex solution for working with Python code in Vim. It has:

- Asynchronous Python code checking (`pylint`, `pyflakes`, `pep8`, `mccabe`) in any combination
- Code refactoring and autocompletion with Rope
- Fast Python folding
- Virtualenv support
- Search through Python documentation and run Python code
- Auto PEP8 error fixes

And more.

SuperTab

SuperTab is a small Vim plugin that makes code completion more convenient by using `<Tab>` key or any other customized keys.

Emacs

Emacs is another powerful text editor. It is fully programmable (lisp), but it can be some work to wire up correctly. A good start if you're already an Emacs user is Python Programming in Emacs at EmacsWiki.

1. Emacs itself comes with a Python mode.

TextMate

> TextMate brings Apple's approach to operating systems into the world of text editors. By bridging UNIX underpinnings and GUI, TextMate cherry-picks the best of both worlds to the benefit of expert scripters and novice users alike.

Sublime Text

> Sublime Text is a sophisticated text editor for code, markup and prose. You'll love the slick user interface, extraordinary features and amazing performance.

Sublime Text has excellent support for editing Python code and uses Python for its plugin API. It also has a diverse variety of plugins, some of which allow for in-editor PEP8 checking and code "linting".

Atom

> Atom is a hackable text editor for the 21st century, built on atom-shell, and based on everything we love about our favorite editors.

Atom is web native (HTML, CSS, JS), focusing on modular design and easy plugin development. It comes with native package control and plethora of packages. Recommended for Python development is Linter combined with linter-flake8.

IDEs

PyCharm / IntelliJ IDEA

PyCharm is developed by JetBrains, also known for IntelliJ IDEA. Both share the same code base and most of PyCharm's features can be brought to IntelliJ with the free Python Plug-In. There are two versions of PyCharm: Professional Edition (Free 30-day trial) and Community Edition (Apache 2.0 License) with fewer features.

Python (on Visual Studio Code)

Python for Visual Studio is an extension for the Visual Studio Code IDE. This is a free, light weight, open source IDE, with support for Mac, Windows, and Linux. Built using open source technologies such as Node.js and Python, with compelling features such as Intellisense (autocompletion), local and remote debugging, linting, and the like.

MIT licensed.

Enthought Canopy

Enthought Canopy is a Python IDE which is focused towards Scientists and Engineers as it provides pre installed libraries for data analysis.

Eclipse

The most popular Eclipse plugin for Python development is Aptana's PyDev.

Komodo IDE

Komodo IDE is developed by ActiveState and is a commercial IDE for Windows, Mac, and Linux. KomodoEdit is the open source alternative.

Spyder

Spyder is an IDE specifically geared toward working with scientific Python libraries (namely Scipy). It includes integration with pyflakes, pylint and rope.

Spyder is open-source (free), offers code completion, syntax highlighting, a class and function browser, and object inspection.

WingIDE

WingIDE is a Python specific IDE. It runs on Linux, Windows and Mac (as an X11 application, which frustrates some Mac users).

WingIDE offers code completion, syntax highlighting, source browser, graphical debugger and support for version control systems.

NINJA-IDE

NINJA-IDE (from the recursive acronym: "Ninja-IDE Is Not Just Another IDE") is a cross-platform IDE, specially designed to build Python applications, and runs on Linux/X11, Mac OS X and Windows desktop operating systems. Installers for these platforms can be downloaded from the website.

NINJA-IDE is open-source software (GPLv3 licence) and is developed in Python and Qt. The source files can be downloaded from GitHub.

Eric (The Eric Python IDE)

Eric is a full featured Python IDE offering sourcecode autocompletion, syntax highlighting, support for version control systems, python 3 support, integrated web browser, python shell, integrated debugger and a flexible plug-in system. Written in python, it is based on the Qt gui toolkit, integrating the Scintilla editor control. Eric is an open-source software project (GPLv3 licence) with more than ten years of active development.

Interpreter Tools

Virtual Environments

Virtual Environments provide a powerful way to isolate project package dependencies. This means that you can use packages particular to a Python project without installing them system wide and thus avoiding potential version conflicts.

To start using and see more information: Virtual Environments docs.

pyenv

pyenv is a tool to allow multiple versions of the Python interpreter to be installed at the same time. This solves the problem of having different projects requiring different versions of Python. For example, it becomes very easy to install Python 2.7 for compatibility in one project, whilst still using Python 3.4 as the default interpreter. pyenv isn't just limited to the CPython versions - it will also install PyPy, anaconda, miniconda, stackless, jython, and ironpython interpreters.

pyenv works by filling a `shims` directory with fake versions of the Python interpreter (plus other tools like `pip` and `2to3`). When the system looks for a program named `python`, it looks inside the `shims` directory first, and uses the

fake version, which in turn passes the command on to pyenv. pyenv then works out which version of Python should be run based on environment variables, `.python-version` files, and the global default.

pyenv isn't a tool for managing virtual environments, but there is the plugin pyenv-virtualenv which automates the creation of different environments, and also makes it possible to use the existing pyenv tools to switch to different environments based on environment variables or `.python-version` files.

Other Tools

IDLE

IDLE is an integrated development environment that is part of Python standard library. It is completely written in Python and uses the Tkinter GUI toolkit. Though IDLE is not suited for full-blown development using Python, it is quite helpful to try out small Python snippets and experiment with different features in Python.

It provides the following features:

- Python Shell Window (interpreter)
- Multi window text editor that colorizes Python code
- Minimal debugging facility

IPython

IPython provides a rich toolkit to help you make the most out of using Python interactively. Its main components are:

- Powerful Python shells (terminal- and Qt-based).
- A web-based notebook with the same core features but support for rich media, text, code, mathematical expressions and inline plots.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into your own projects.
- Tools for high level and interactive parallel computing.

```
$ pip install ipython
```

To download and install IPython with all it's optional dependencies for the notebook, qtconsole, tests, and other functionalities

```
$ pip install ipython[all]
```

BPython

bpython is an alternative interface to the Python interpreter for Unix-like operating systems. It has the following features:

- In-line syntax highlighting.
- Readline-like autocomplete with suggestions displayed as you type.
- Expected parameter list for any Python function.
- "Rewind" function to pop the last line of code from memory and re-evaluate.
- Send entered code off to a pastebin.

- Save entered code to a file.
- Auto-indentation.
- Python 3 support.

```
$ pip install bpython
```

ptpython

ptpython is a REPL build on top of the prompt_toolkit library. It is considered to be an alternative to BPython. Features include:

- Syntax highlighting
- Autocompletion
- Multiline editing
- Emacs and VIM Mode
- Embedding REPL inside of your code
- Syntax Validation
- Tab pages
- Support for integrating with IPython's shell, by installing IPython `pip install ipython` and running `ptipython`.

```
$ pip install ptpython
```

Virtual Environments

A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the "Project X depends on version 1.x but, Project Y needs 4.x" dilemma, and keeps your global site-packages directory clean and manageable.

For example, you can work on a project which requires Django 1.10 while also maintaining a project which requires Django 1.8.

virtualenv

virtualenv is a tool to create isolated Python environments. virtualenv creates a folder which contains all the necessary executables to use the packages that a Python project would need.

Install virtualenv via pip:

```
$ pip install virtualenv
```

Basic Usage

1. Create a virtual environment for a project:

```
$ cd my_project_folder
$ virtualenv venv
```

`virtualenv venv` will create a folder in the current directory which will contain the Python executable files, and a copy of the `pip` library which you can use to install other packages. The name of the virtual environment (in this case, it was `venv`) can be anything; omitting the name will place the files in the current directory instead.

This creates a copy of Python in whichever directory you ran the command in, placing it in a folder named `venv`.

You can also use the Python interpreter of your choice (like `python2.7`).

```
$ virtualenv -p /usr/bin/python2.7 venv
```

or change the interpreter globally with an env variable in `~/.bashrc`:

```
$ export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python2.7
```

2. To begin using the virtual environment, it needs to be activated:

```
$ source venv/bin/activate
```

The name of the current virtual environment will now appear on the left of the prompt (e.g. `(venv)Your-Computer:your_project UserName$`) to let you know that it's active. From now on, any package that you install using pip will be placed in the `venv` folder, isolated from the global Python installation.

Install packages as usual, for example:

```
$ pip install requests
```

3. If you are done working in the virtual environment for the moment, you can deactivate it:

```
$ deactivate
```

This puts you back to the system's default Python interpreter with all its installed libraries.

To delete a virtual environment, just delete its folder. (In this case, it would be `rm -rf venv`.)

After a while, though, you might end up with a lot of virtual environments littered across your system, and its possible you'll forget their names or where they were placed.

Other Notes

Running `virtualenv` with the option `--no-site-packages` will not include the packages that are installed globally. This can be useful for keeping the package list clean in case it needs to be accessed later. [This is the default behavior for `virtualenv` 1.7 and later.]

In order to keep your environment consistent, it's a good idea to "freeze" the current state of the environment packages. To do this, run

```
$ pip freeze > requirements.txt
```

This will create a `requirements.txt` file, which contains a simple list of all the packages in the current environment, and their respective versions. You can see the list of installed packages without the requirements format using "pip list". Later it will be easier for a different developer (or you, if you need to re-create the environment) to install the same packages using the same versions:

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

This can help ensure consistency across installations, across deployments, and across developers.

Lastly, remember to exclude the virtual environment folder from source control by adding it to the ignore list.

virtualenvwrapper

virtualenvwrapper provides a set of commands which makes working with virtual environments much more pleasant. It also places all your virtual environments in one place.

To install (make sure **virtualenv** is already installed):

```
$ pip install virtualenvwrapper
$ export WORKON_HOME=~/Envs
$ source /usr/local/bin/virtualenvwrapper.sh
```

(Full virtualenvwrapper install instructions.)

For Windows, you can use the virtualenvwrapper-win.

To install (make sure **virtualenv** is already installed):

```
$ pip install virtualenvwrapper-win
```

In Windows, the default path for WORKON_HOME is %USERPROFILE%Envs

Basic Usage

1. Create a virtual environment:

```
$ mkvirtualenv venv
```

This creates the `venv` folder inside `~/Envs`.

2. Work on a virtual environment:

```
$ workon venv
```

Alternatively, you can make a project, which creates the virtual environment, and also a project directory inside `$PROJECT_HOME`, which is `cd` -ed into when you `workon myproject`.

```
$ mkproject myproject
```

virtualenvwrapper provides tab-completion on environment names. It really helps when you have a lot of environments and have trouble remembering their names.

`workon` also deactivates whatever environment you are currently in, so you can quickly switch between environments.

3. Deactivating is still the same:

```
$ deactivate
```

4. To delete:

```
$ rmvirtualenv venv
```

Other useful commands

`lsvirtualenv` List all of the environments.

`cdvirtualenv` Navigate into the directory of the currently activated virtual environment, so you can browse its `site-packages`, for example.

`cdsitepackages` Like the above, but directly into `site-packages` directory.

`lssitepackages` Shows contents of `site-packages` directory.

Full list of virtualenvwrapper commands.

virtualenv-burrito

With virtualenv-burrito, you can have a working virtualenv + virtualenvwrapper environment in a single command.

autoenv

When you `cd` into a directory containing a `.env`, autoenv automagically activates the environment.

Install it on Mac OS X using `brew`:

```
$ brew install autoenv
```

And on Linux:

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

Further Configuration of Pip and Virtualenv

Requiring an active virtual environment for `pip`

By now it should be clear that using virtual environments is a great way to keep your development environment clean and keeping different projects' requirements separate.

When you start working on many different projects, it can be hard to remember to activate the related virtual environment when you come back to a specific project. As a result of this, it is very easy to install packages globally while thinking that you are actually installing the package for the virtual environment of the project. Over time this can result in a messy global package list.

In order to make sure that you install packages to your active virtual environment when you use `pip install`, consider adding the following line to your `~/.bashrc` file:

```
export PIP_REQUIRE_VIRTUALENV=true
```

After saving this change and sourcing the `~/.bashrc` file with `source ~/.bashrc`, pip will no longer let you install packages if you are not in a virtual environment. If you try to use `pip install` outside of a virtual environment pip will gently remind you that an activated virtual environment is needed to install packages.

```
$ pip install requests
Could not find an activated virtualenv (required).
```

You can also do this configuration by editing your `pip.conf` or `pip.ini` file. `pip.conf` is used by Unix and Mac OS X operating systems and it can be found at:

```
$HOME/.pip/pip.conf
```

Similarly, the `pip.ini` file is used by Windows operating systems and it can be found at:

```
%HOME%\pip\pip.ini
```

If you don't have a `pip.conf` or `pip.ini` file at these locations, you can create a new file with the correct name for your operating system.

If you already have a configuration file, just add the following line under the `[global]` settings to require an active virtual environment:

```
require-virtualenv = true
```

If you did not have a configuration file, you will need to create a new one and add the following lines to this new file:

```
[global]
require-virtualenv = true
```

You will of course need to install some packages globally (usually ones that you use across different projects consistently) and this can be accomplished by adding the following to your `~/.bashrc` file:

```
gpip() {
    PIP_REQUIRE_VIRTUALENV="" pip "$@"
}
```

After saving the changes and sourcing your `~/.bashrc` file you can now install packages globally by running `gpip install`. You can change the name of the function to anything you like, just keep in mind that you will have to use that name when trying to install packages globally with pip.

Caching packages for future use

Every developer has preferred libraries and when you are working on a lot of different projects, you are bound to have some overlap between the libraries that you use. For example, you may be using the `requests` library in a lot of different projects.

It is surely unnecessary to re-download the same packages/libraries each time you start working on a new project (and in a new virtual environment as a result). Fortunately, you can configure pip in such a way that it tries to reuse already installed packages.

On UNIX systems, you can add the following line to your `.bashrc` or `.bash_profile` file.

```
export PIP_DOWNLOAD_CACHE=$HOME/.pip/cache
```

You can set the path to anywhere you like (as long as you have write access). After adding this line, `source` your `.bashrc` (or `.bash_profile`) file and you will be all set.

Another way of doing the same configuration is via the `pip.conf` or `pip.ini` files, depending on your system. If you are on Windows, you can add the following line to your `pip.ini` file under `[global]` settings:

```
download-cache = %HOME%\pip\cache
```

Similarly, on UNIX systems you should simply add the following line to your `pip.conf` file under `[global]` settings:

```
download-cache = $HOME/.pip/cache
```

Even though you can use any path you like to store your cache, it is recommended that you create a new folder *in* the folder where your `pip.conf` or `pip.ini` file lives. If you don't trust yourself with all of this path voodoo, just use the values provided here and you will be fine.

CHAPTER 6

Additional Notes

This part of the guide, which is mostly prose, begins with some background information about Python, then focuses on next steps.

Introduction

From the official Python website:

Python is a general-purpose, high-level programming language similar to Tcl, Perl, Ruby, Scheme, or Java. Some of its main key features include:

- **very clear, readable syntax**

  Python's philosophy focuses on readability, from code blocks delineated with significant whitespace to intuitive keywords in place of inscrutable punctuation.

- **extensive standard libraries and third party modules for virtually any task**

  Python is sometimes described with the words "batteries included" because of its extensive standard library, which includes modules for regular expressions, file IO, fraction handling, object serialization, and much more.

  Additionally, the Python Package Index is available for users to submit their packages for widespread use, similar to Perl's CPAN. There is a thriving community of very powerful Python frameworks and tools like the Django web framework and the NumPy set of math routines.

- **integration with other systems**

  Python can integrate with Java libraries, enabling it to be used with the rich Java environment that corporate programmers are used to. It can also be extended by C or C++ modules when speed is of the essence.

- **ubiquity on computers**

  Python is available on Windows, *nix, and Mac. It runs wherever the Java virtual machine runs, and the reference implementation CPython can help bring Python to wherever there is a working C compiler.

- **friendly community**

  Python has a vibrant and large *community* which maintains wikis, conferences, countless repositories, mailing lists, IRC channels, and so much more. Heck, the Python community is even helping to write this guide!

About This Guide

Purpose

The Hitchhiker's Guide to Python exists to provide both novice and expert Python developers a best practice handbook for the installation, configuration, and usage of Python on a daily basis.

By the Community

This guide is architected and maintained by Kenneth Reitz in an open fashion. This is a community-driven effort that serves one purpose: to serve the community.

For the Community

All contributions to the Guide are welcome, from Pythonistas of all levels. If you think there's a gap in what the Guide covers, fork the Guide on GitHub and submit a pull request.

Contributions are welcome from everyone, whether they're an old hand or a first-time Pythonista, and the authors to the Guide will gladly help if you have any questions about the appropriateness, completeness, or accuracy of a contribution.

To get started working on The Hitchhiker's Guide, see the Contribute page.

The Community

BDFL

Guido van Rossum, the creator of Python, is often referred to as the BDFL — the Benevolent Dictator For Life.

Python Software Foundation

The mission of the Python Software Foundation is to promote, protect, and advance the Python programming language, and to support and facilitate the growth of a diverse and international community of Python programmers.

Learn More about the PSF.

PEPs

PEPs are *Python Enhancement Proposals*. They describe changes to Python itself, or the standards around it.

There are three different types of PEPs (as defined by **PEP 1**):

Standards Describes a new feature or implementation.

Informational Describes a design issue, general guidelines, or information to the community.

Process Describes a process related to Python.

Notable PEPs

There are a few PEPs that could be considered required reading:

- **PEP 8: The Python Style Guide.** Read this. All of it. Follow it.
- **PEP 20: The Zen of Python.** A list of 19 statements that briefly explain the philosophy behind Python.

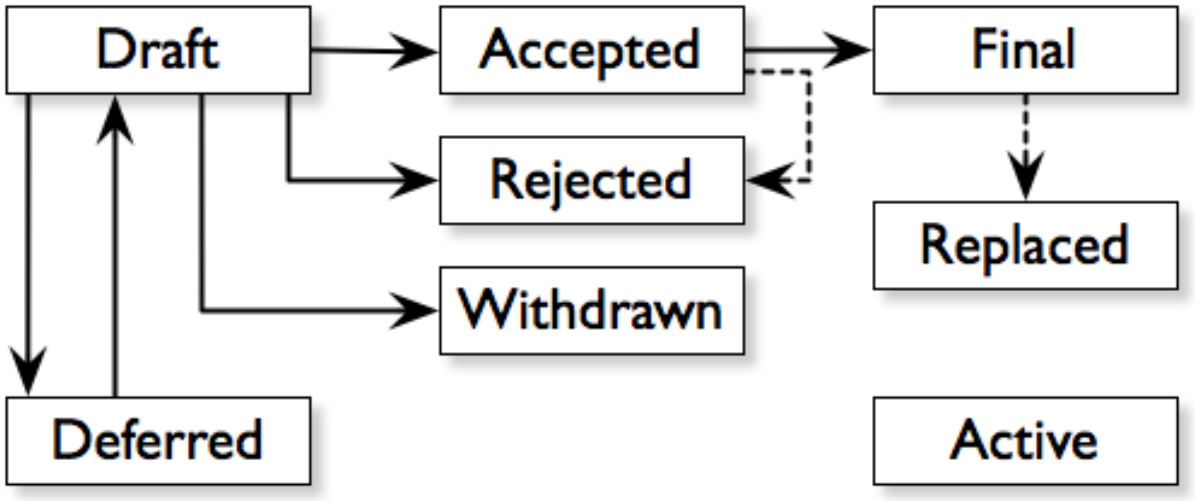
- **PEP 257: Docstring Conventions.** Gives guidelines for semantics and conventions associated with Python docstrings.

You can read more at The PEP Index.

Submitting a PEP

PEPs are peer-reviewed and accepted/rejected after much discussion. Anyone can write and submit a PEP for review.

Here's an overview of the PEP acceptance workflow:



Python Conferences

The major events for the Python community are developer conferences. The two most notable conferences are PyCon, which is held in the US, and its European sibling, EuroPython.

A comprehensive list of conferences is maintained at pycon.org.

Python User Groups

User Groups are where a bunch of Python developers meet to present or talk about Python topics of interest. A list of local user groups is maintained at the Python Software Foundation Wiki.

Learning Python

Beginner

The Python Tutorial

This is the official tutorial. It covers all the basics, and offers a tour of the language and the standard library. Recommended for those who need a quick-start guide to the language.

The Python Tutorial

Python for Beginners

thepythonguru.com is a tutorial focuses on beginner programmers. It covers many python concepts in depth. It also teaches you some advance constructs of python like lambda expression, regular expression. At last it finishes off with tutorial "How to access MySQL db using python"

Python for beginners

Learn Python Interactive Tutorial

Learnpython.org is an easy non-intimidating way to get introduced to Python. The website takes the same approach used on the popular Try Ruby website, it has an interactive Python interpreter built into the site that allows you to go through the lessons without having to install Python locally.

Learn Python

If you want a more traditional book, *Python For You and Me* is an excellent resource for learning all aspects of the language.

Python for You and Me

Online Python Tutor

Online Python Tutor gives you a visual step by step representation of how your program runs. Python Tutor helps people overcome a fundamental barrier to learning programming by understanding what happens as the computer executes each line of a program's source code.

Online Python Tutor

Invent Your Own Computer Games with Python

This beginner's book is for those with no programming experience at all. Each chapter has the source code to a small game, using these example programs to demonstrate programming concepts to give the reader an idea of what programs "look like".

Invent Your Own Computer Games with Python

Hacking Secret Ciphers with Python

This book teaches Python programming and basic cryptography for absolute beginners. The chapters provide the source code for various ciphers, as well as programs that can break them.

Hacking Secret Ciphers with Python

Learn Python the Hard Way

This is an excellent beginner programmer's guide to Python. It covers "hello world" from the console to the web.

Learn Python the Hard Way

Crash into Python

Also known as *Python for Programmers with 3 Hours*, this guide gives experienced developers from other languages a crash course on Python.

Crash into Python

Dive Into Python 3

Dive Into Python 3 is a good book for those ready to jump in to Python 3. It's a good read if you are moving from Python 2 to 3 or if you already have some experience programming in another language.

Dive Into Python 3

Think Python: How to Think Like a Computer Scientist

Think Python attempts to give an introduction to basic concepts in computer science through the use of the Python language. The focus was to create a book with plenty of exercises, minimal jargon and a section in each chapter devoted to the subject of debugging.

While exploring the various features available in the Python language the author weaves in various design patterns and best practices.

The book also includes several case studies which have the reader explore the topics discussed in the book in greater detail by applying those topics to real-world examples. Case studies include assignments in GUI and Markov Analysis.

Think Python

Python Koans

Python Koans is a port of Edgecase's Ruby Koans. It uses a test-driven approach, q.v. TEST DRIVEN DESIGN SECTION to provide an interactive tutorial teaching basic Python concepts. By fixing assertion statements that fail in a test script, this provides sequential steps to learning Python.

For those used to languages and figuring out puzzles on their own, this can be a fun, attractive option. For those new to Python and programming, having an additional resource or reference will be helpful.

Python Koans

More information about test driven development can be found at these resources:

Test Driven Development

A Byte of Python

A free introductory book that teaches Python at the beginner level, it assumes no previous programming experience.

A Byte of Python for Python 2.x A Byte of Python for Python 3.x

Learn to Program in Python with Codeacademy

A Codeacademy course for the absolute Python beginner. This free and interactive course provides and teaches the basics (and beyond) of Python programming whilst testing the user's knowledge in between progress. This course also features a built-in interpreter for receiving instant feedback on your learning.

Learn to Program in Python with Codeacademy

Intermediate

Effective Python

This book contains 59 specific ways to improve writing Pythonic code. At 227 pages, it is a very brief overview of some of the most commons adapations programmers need to make to become efficient intermediate level Python programmers.

Effective Python

Advanced

Pro Python

This book is for intermediate to advanced Python programmers who are looking to understand how and why Python works the way it does and how they can take their code to the next level.

Pro Python

Expert Python Programming

Expert Python Programming deals with best practices in programming Python and is focused on the more advanced crowd.

It starts with topics like decorators (with caching, proxy, and context manager case-studies), method resolution order, using super() and meta-programming, and general **PEP 8** best practices.

It has a detailed, multi-chapter case study on writing and releasing a package and eventually an application, including a chapter on using zc.buildout. Later chapters detail best practices such as writing documentation, test-driven development, version control, optimization and profiling.

Expert Python Programming

A Guide to Python's Magic Methods

This is a collection of blog posts by Rafe Kettler which explain 'magic methods' in Python. Magic methods are surrounded by double underscores (i.e. __init__) and can make classes and objects behave in different and magical ways.

A Guide to Python's Magic Methods

Note: The Rafekettler.com is currently down, you can go to their Github version directly. Here you can find a PDF version: A Guide to Python's Magic Methods (repo on GitHub)

For Engineers and Scientists

A Primer on Scientific Programming with Python

A Primer on Scientific Programming with Python, written by Hans Petter Langtangen, mainly covers Python's usage in the scientific field. In the book, examples are chosen from mathematics and the natural sciences.

A Primer on Scientific Programming with Python

Numerical Methods in Engineering with Python

Numerical Methods in Engineering with Python, written by Jaan Kiusalaas, puts the emphasis on numerical methods and how to implement them in Python.

Numerical Methods in Engineering with Python

Miscellaneous topics

Problem Solving with Algorithms and Data Structures

Problem Solving with Algorithms and Data Structures covers a range of data structures and algorithms. All concepts are illustrated with Python code along with interactive samples that can be run directly in the browser.

Problem Solving with Algorithms and Data Structures

Programming Collective Intelligence

Programming Collective Intelligence introduces a wide array of basic machine learning and data mining methods. The exposition is not very mathematically formal, but rather focuses on explaining the underlying intuition and shows how to implement the algorithms in Python.

Programming Collective Intelligence

Transforming Code into Beautiful, Idiomatic Python

Transforming Code into Beautiful, Idiomatic Python is a video by Raymond Hettinger. Learn to take better advantage of Python's best features and improve existing code through a series of code transformations, "When you see this, do that instead."

Transforming Code into Beautiful, Idiomatic Python

Fullstack Python

Fullstack Python offers a complete top-to-bottom resource for web development using Python.

From setting up the webserver, to designing the front-end, choosing a database, optimizing/scaling, etc.

As the name suggests, it covers everything you need to build and run a complete web app from scratch.

Fullstack Python

References

Python in a Nutshell

Python in a Nutshell, written by Alex Martelli, covers most cross-platform Python's usage, from its syntax to built-in libraries to advanced topics such as writing C extensions.

Python in a Nutshell

The Python Language Reference

This is Python's reference manual, it covers the syntax and the core semantics of the language.

The Python Language Reference

Python Essential Reference

Python Essential Reference, written by David Beazley, is the definitive reference guide to Python. It concisely explains both the core language and the most essential parts of the standard library. It covers Python 3 and 2.6 versions.

Python Essential Reference

Python Pocket Reference

Python Pocket Reference, written by Mark Lutz, is an easy to use reference to the core language, with descriptions of commonly used modules and toolkits. It covers Python 3 and 2.6 versions.

Python Pocket Reference

Python Cookbook

Python Cookbook, written by David Beazley and Brian K. Jones, is packed with practical recipes. This book covers the core python language as well as tasks common to a wide variety of application domains.

Python Cookbook

Writing Idiomatic Python

"Writing Idiomatic Python", written by Jeff Knupp, contains the most common and important Python idioms in a format that maximizes identification and understanding. Each idiom is presented as a recommendation of a way to write some commonly used piece of code, followed by an explanation of why the idiom is important. It also contains two code samples for each idiom: the "Harmful" way to write it and the "Idiomatic" way.

For Python 2.7.3+

For Python 3.3+

Documentation

Official Documentation

The official Python Language and Library documentation can be found here:

- Python 2.x
- Python 3.x

Read the Docs

Read the Docs is a popular community project that hosts documentation for open source software. It holds documentation for many Python modules, both popular and exotic.

Read the Docs

pydoc

pydoc is a utility that is installed when you install Python. It allows you to quickly retrieve and search for documentation from your shell. For example, if you needed a quick refresher on the `time` module, pulling up documentation would be as simple as

```
$ pydoc time
```

The above command is essentially equivalent to opening the Python REPL and running

```
>>> help(time)
```

News

Planet Python

This is an aggregate of Python news from a growing number of developers.

Planet Python

/r/python

/r/python is the Reddit Python community where users contribute and vote on Python-related news.

/r/python

Pycoder's Weekly

Pycoder's Weekly is a free weekly Python newsletter for Python developers by Python developers (Projects, Articles, News, and Jobs).

Pycoder's Weekly

Python Weekly

Python Weekly is a free weekly newsletter featuring curated news, articles, new releases, jobs, etc. related to Python.

Python Weekly

Python News

Python News is the news section in the official Python web site (www.python.org). It briefly highlights the news from the Python community.

Python News

Import Python Weekly

Weekly Python Newsletter containing Python Articles, Projects, Videos, Tweets delivered in your inbox. Keep Your Python Programming Skills Updated.

Import Python Weekly Newsletter

Awesome Python Newsletter

A weekly overview of the most popular Python news, articles and packages.

Awesome Python Newsletter

Note: Notes defined within all diatonic and chromatic musical scales have been intentionally excluded from this list of additional notes. Additionally, this note.

Contribution notes and legal information (for those interested).

Contribute

Python-guide is under active development, and contributors are welcome.

If you have a feature request, suggestion, or bug report, please open a new issue on GitHub. To submit patches, please send a pull request on GitHub. Once your changes get merged back in, you'll automatically be added to the Contributors List.

Style Guide

For all contributions, please follow the *The Guide Style Guide*.

Todo List

If you'd like to contribute, there's plenty to do. Here's a short todo list.

- Establish "use this" vs "alternatives are...." recommendations

---

Todo

Write about Blueprint

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-guide/checkouts/latest/docs/scenarios/admin.rst, line 369.)

Todo

Fill in "Freezing Your Code" stub

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-guide/checkouts/latest/docs/shipping/freezing.rst, line 37.)

Todo

Write steps for most basic .exe

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-guide/checkouts/latest/docs/shipping/freezing.rst, line 73.)

Todo

Include code examples of exemplary code from each of the projects listed. Explain why it is excellent code. Use complex examples.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-guide/checkouts/latest/docs/writing/reading.rst, line 57.)

Todo

Explain techniques to rapidly identify data structures, algorithms and determine what the code is doing.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-guide/checkouts/latest/docs/writing/reading.rst, line 59.)

License

The Guide Style Guide

As with all documentation, having a consistent format helps make the document more understandable. In order to make The Guide easier to digest, all contributions should fit within the rules of this style guide where appropriate.

The Guide is written as *reStructuredText*.

Note: Parts of The Guide may not yet match this style guide. Feel free to update those parts to be in sync with The Guide Style Guide

Note: On any page of the rendered HTML you can click "Show Source" to see how authors have styled the page.

Relevancy

Strive to keep any contributions relevant to the *purpose of The Guide*.

- Avoid including too much information on subjects that don't directly relate to Python development.

- Prefer to link to other sources if the information is already out there. Be sure to describe what and why you are linking.
- Cite references where needed.
- If a subject isn't directly relevant to Python, but useful in conjunction with Python (e.g., Git, GitHub, Databases), reference by linking to useful resources, and describe why it's useful to Python.
- When in doubt, ask.

Headings

Use the following styles for headings.

Chapter title:

```
#########
Chapter 1
#########
```

Page title:

```
====================
Time is an Illusion
====================
```

Section headings:

```
Lunchtime Doubly So
-------------------
```

Sub section headings:

```
Very Deep
~~~~~~~~~
```

Prose

Wrap text lines at 78 characters. Where necessary, lines may exceed 78 characters, especially if wrapping would make the source text more difficult to read.

Use of the serial comma (also known as the Oxford comma) is 100% non-optional. Any attempt to submit content with a missing serial comma will result in permanent banishment from this project, due to complete and total lack of taste.

Banishment? Is this a joke? Hopefully we will never have to find out.

Code Examples

Wrap all code examples at 70 characters to avoid horizontal scrollbars.

Command line examples:

```
.. code-block:: console

    $ run command --help
    $ ls ..
```

Be sure to include the `$` prefix before each line.

Python interpreter examples:

```
Label the example::

.. code-block:: python

    >>> import this
```

Python examples:

```
Descriptive title::

.. code-block:: python

    def get_answer():
        return 42
```

Externally Linking

- Prefer labels for well known subjects (ex: proper nouns) when linking:

```
    Sphinx_ is used to document Python.

    .. _Sphinx: http://sphinx.pocoo.org
```

- Prefer to use descriptive labels with inline links instead of leaving bare links:

```
    Read the `Sphinx Tutorial <http://sphinx.pocoo.org/tutorial.html>`_
```

- Avoid using labels such as "click here", "this", etc. preferring descriptive labels (SEO worthy) instead.

Linking to Sections in The Guide

To cross-reference other parts of this documentation, use the :ref: keyword and labels.

To make reference labels more clear and unique, always add a `-ref` suffix:

```
.. _some-section-ref:

Some Section
------------
```

Notes and Warnings

Make use of the appropriate admonitions directives when making notes.

Notes:

```
.. note::
    The Hitchhiker's Guide to the Galaxy has a few things to say
    on the subject of towels. A towel, it says, is about the most
    massively useful thing an interstellar hitch hiker can have.
```

Warnings:

```
.. warning:: DON'T PANIC
```

TODOs

Please mark any incomplete areas of The Guide with a todo directive. To avoid cluttering the *Todo List*, use a single `todo` for stub documents or large incomplete sections.

```
.. todo::
    Learn the Ultimate Answer to the Ultimate Question
    of Life, The Universe, and Everything
```

E

environment variable
    PATH, 6, 7

P

PATH, 6, 7
Python Enhancement Proposals
    PEP 0257#specification, 33
    PEP 1, 102
    PEP 20, 27, 102
    PEP 249, 59
    PEP 257, 35, 103
    PEP 282, 39
    PEP 3101, 21
    PEP 3132, 25
    PEP 3333, 48
    PEP 391, 41
    PEP 8, 28, 91, 102, 106
    PEP 8#comments, 33