Python Guide Documentation

Release 0.0.1

Kenneth Reitz

fev 07, 2022

Sumário

Saudações, Terráqueos! Bem-vindos ao Guia do Mochileiro para 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 for the installation, configuration, and usage of Python on a daily basis.

Esse guia é realizado de uma forma que é quase, mas não 100%, totalmente diferente da documentação oficial do Python. Você não vai achar uma lista de todos os frameworks web disponíveis para Python aqui: você achará uma lista concisa de opções altamente recomendadas.

Nota: The use of **Python 3** is *highly* recommended over Python 2. Consider upgrading your applications and infrastructures if you find yourself *still* using Python 2 in production today. If you are using Python 3, congratulations — you are indeed a person of excellent taste. —*Kenneth Reitz*

Vamos começar! Mas primeiro vamos ter certeza de que você sabe onde está sua toalha.

CAPÍTULO 1

Começando com Python

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

1.1 Picking a Python Interpreter (3 vs 2)

1.1.1 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 3.
2. Python 3 is ready for the production deployment of applications today.
3. Python 2 reached the end of its life on January 1, 2020[6].
4. The brand name "Python" encapsulates both Python 3 and Python 2.

1.1.2 Recommendations

Nota: The use of **Python 3** is *highly* recommended over Python 2. Consider upgrading your applications and infrastructure if you find yourself *still* using Python 2 in production today. If you are using Python 3, congratulations — you are indeed a person of excellent taste. —*Kenneth Reitz*

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".

1.1.3 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.

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.

1.1.4 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.

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

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]

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] IronPython 3[4] is being developed, but is not ready for use as of September 2020.

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 is compatible with Python 2.7 and 3.5-3.8.[5]

- Properly Install Python on your system:

[1] https://pypy.org/compat.html
[2] https://hg.python.org/jython/file/412a8f9445f7/NEWS
[3] https://ironpython.net/download/
[4] https://github.com/IronLanguages/ironpython3
[5] https://pythonnet.github.io/

1.2 Instalando Python corretamente.

Há uma boa chance the que você já tenha Python instalado em seu sistema operacional.

Se for o caso, você não precisa instalar ou configurar mais nada para usar Python. Dito isso, eu recomendo fortemente que você instale as ferramentas e bibliotecas descritas nos guias abaixo antes de construir seus aplicativos em Python para usar no mundo real. Em particular, você deve sempre instalar **Setuptools**, **Pip** e **Virtualenv** - essas ferramentas facilitam muito o uso de outras bibliotecas de Python.

Nota: The use of **Python 3** is *highly* preferred over Python 2. Consider upgrading your applications and infrastructure if you find yourself *still* using Python 2 in production today. If you are using Python 3, congratulations — you are indeed a person of excellent taste. —*Kenneth Reitz*

1.2.1 Guias de Instalação

Esses guias cobre a instalação de *Python* para desenvolviment, bem como o **setuptools**, **Pip** e **virtualenv**.

Python 3 Installation Guides

- *Python 3 no MacOS*.
- *Python 3 on Windows*.
- *Python 3 on Linux*.

Legacy Python 2 Installation Guides

- *Python 2 no MacOS*.
- *Python 2 no Microsoft Windows*.
- *Python 2 on Linux*.

1.3 Instalando Pyhton 3 no Mac OS X

Mac OS X comes with Python 2.7 out of the box.

You do not need to install or configure anything else to use Python 2. These instructions document the installation of Python 3.

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.

1.3.1 Doing it Right

Let's install a real version of Python.

Before installing Python, you'll need to install GCC. GCC can be obtained by downloading Xcode, the smaller Command Line Tools (must have an Apple account) or the even smaller OSX-GCC-Installer package.

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

Nota: 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 package manager. Homebrew fills this void.

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

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
↪install/master/install.sh)"
```

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/opt/python/libexec/bin:$PATH"
```

If you have OS X 10.12 (Sierra) or older use this line instead

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

Now, we can install Python 3:

```
$ brew install python
```

This will take a minute or two.

1.3.2 Pip

Homebrew installs `pip` pointing to the Homebrew'd Python 3 for you.

1.3.3 Working with Python 3

At this point, you have the system Python 2.7 available, potentially the *Homebrew version of Python 2* installed, and the Homebrew version of Python 3 as well.

```
$ python
```

will launch the Homebrew-installed Python 3 interpreter.

```
$ python2
```

will launch the Homebrew-installed Python 2 interpreter (if any).

```
$ python3
```

will launch the Homebrew-installed Python 3 interpreter.

If the Homebrew version of Python 2 is installed then `pip2` will point to Python 2. If the Homebrew version of Python 3 is installed then `pip` will point to Python 3.

The rest of the guide will assume that `python` references Python 3.

```
# Do I have a Python 3 installed?
$ python --version
Python 3.7.1 # Success!
```

1.3.4 Pipenv & Virtual Environments

The next step is to install Pipenv, so you can install dependencies and manage virtual environments.

Um Virtual Environment (Ambiente virtual) é uma ferramenta que permite guardar as dependências de projetos diferentes em lugares separados criando um ambiente virtual Python para cada um deles. Isso resolve problemas como "O projeto X usa uma biblioteca na versão 1.x mas o projeto Y usa essa mesma biblioteca mas na versão 4.x" e mantém os seus pacotes instalados na pasta site-packages global limpa e organizada.

Por exemplo, você pode trabalhar em um projeto que usa o Django na versão 1.10 enquanto também poderá trabalhar em um outro projeto que use o Django mas na versão 1.8.

So, onward! To the *Pipenv & Virtual Environments* docs!

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

1.4 Installing Python 3 on Windows

First, follow the installation instructions for Chocolatey. It's a community system packager manager for Windows 7+. (It's very much like Homebrew on OS X.)

Once done, installing Python 3 is very simple, because Chocolatey pushes Python 3 as the default.

```
choco install python
```

Once you've run this command, you should be able to launch Python directly from to the console. (Chocolatey is fantastic and automatically adds Python to your path.)

1.4.1 Setuptools + Pip

The two most crucial third-party Python packages are setuptools and pip, which let you 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.

All supported versions of Python 3 include pip, so just make sure it's up to date:

```
python -m pip install -U pip
```

1.4.2 Pipenv & Virtual Environments

The next step is to install Pipenv, so you can install dependencies and manage 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 2.0 while also maintaining a project which requires Django 1.8.

So, onward! To the *Pipenv & Virtual Environments* docs!

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

1.5 Installing Python 3 on Linux

This document describes how to install Python 3.6 or 3.8 on Ubuntu Linux machines.

To see which version of Python 3 you have installed, open a command prompt and run

```
$ python3 --version
```

If you are using Ubuntu 16.10 or newer, then you can easily install Python 3.6 with the following commands:

```
$ sudo apt-get update
$ sudo apt-get install python3.6
```

If you're using another version of Ubuntu (e.g. the latest LTS release) or you want to use a more current Python, we recommend using the deadsnakes PPA to install Python 3.8:

```
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install python3.8
```

If you are using other Linux distribution, chances are you already have Python 3 pre-installed as well. If not, use your distribution's package manager. For example on Fedora, you would use *dnf*:

```
$ sudo dnf install python3
```

Note that if the version of the `python3` package is not recent enough for you, there may be ways of installing more recent versions as well, depending on you distribution. For example installing the `python3.9` package on Fedora 32 to get Python 3.9. If you are a Fedora user, you might want to read about multiple Python versions available in Fedora.

1.5.1 Working with Python 3

At this point, you may have system Python 2.7 available as well.

```
$ python
```

This might launch the Python 2 interpreter.

```
$ python3
```

This will always launch the Python 3 interpreter.

1.5.2 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.

Note that on some Linux distributions including Ubuntu and Fedora the `pip` command is meant for Python 2, while the `pip3` command is meant for Python 3.

```
$ command -v pip3
```

However, when using virtual environments (described below), you don't need to care about that.

1.5.3 Pipenv & Virtual Environments

The next step is to install Pipenv, so you can install dependencies and manage 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.

So, onward! To the *Pipenv & Virtual Environments* docs!

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

1.6 Installing Python 2 on Mac OS X

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

Mac OS X 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.

1.6.1 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.

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

Nota: 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 OS X terminal emulator and run

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

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@2
```

Because `python@2` is a "keg", we need to update our `PATH` again, to point at our new installation:

```
export PATH="/usr/local/opt/python@2/libexec/bin:$PATH"
```

Homebrew names the executable `python2` so that you can still run the system Python via the executable `python`.

```
$ python -V   # Homebrew installed Python 3 interpreter (if installed)
$ python2 -V  # Homebrew installed Python 2 interpreter
$ python3 -V  # Homebrew installed Python 3 interpreter (if installed)
```

1.6.2 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.

```
$ pip2 -V  # pip pointing to the Homebrew installed Python 2 interpreter
$ pip -V  # pip pointing to the Homebrew installed Python 3 interpreter (if␣
↪installed)
```

1.6.3 Ambientes virtuais

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.

Por exemplo, você pode trabalhar em um projeto que usa o Django na versão 1.10 enquanto também poderá trabalhar em um outro projeto que use o Django mas na versão 1.8.

Para começar a usar isso e ver mais informações: documentação *Ambientes Virtuais*.

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

1.7 Installing Python 2 on Windows

Nota: Check out our *guide for installing Python 3 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\", "User")
```

This is also an option during the installation process.

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.

1.7.1 Setuptools + Pip

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

Uma vez instalado, você pode baixar, instalar e desinstalar qualquer produto de software Python compatível com um único comando. Ele também permite que você adicione esse recurso de instalação de rede ao seu próprio software Python com muito pouco trabalho.

O Python 2.7.9 e posterior (na série python2) e o Python 3.4 e posterior incluem pip por padrão.

Para ver se o pip está instalado, abra um prompt de comando e execute

```
command -v pip
```

Para instalar o pip, siga o guia oficial de instalação do pip - isso instalará automaticamente a versão mais recente do setuptools.

1.7.2 Ambientes virtuais

Um Virtual Environment (Ambiente virtual) é uma ferramenta que permite guardar as dependências de projetos diferentes em lugares separados criando um ambiente virtual Python para cada um deles. Isso resolve problemas como "O projeto X usa uma biblioteca na versão 1.x mas o projeto Y usa essa mesma biblioteca mas na versão 4.x" e mantém os seus pacotes instalados na pasta site-packages global limpa e organizada.

Por exemplo, você pode trabalhar em um projeto que usa o Django na versão 1.10 enquanto também poderá trabalhar em um outro projeto que use o Django mas na versão 1.8.

Para começar a usar isso e ver mais informações: documentação *Ambientes Virtuais*.

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

1.8 Installing Python 2 on Linux

Nota: Check out our *guide for installing Python 3 on Linux*.

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

Para ver qual versão do Python você instalou, abra um prompt de comando e execute

```
$ python2 --version
```

However, with the growing popularity of Python 3, some distributions, such as Fedora, don't come with Python 2 pre-installed. You can install the `python2` package with your distribution package manager:

```
$ sudo dnf install python2
```

Você não precisa instalar ou configurar qualquer outra coisa para usar o Python. Tendo dito isso, eu recomendo fortemente que você instale as ferramentas e bibliotecas descritas na próxima seção antes de começar a construir aplicativos Python para uso no mundo real. Em particular, você deve sempre instalar o Setuptools e o pip, já que isso facilita muito o uso de outras bibliotecas Python de terceiros.

1.8.1 Setuptools & Pip

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

Uma vez instalado, você pode baixar, instalar e desinstalar qualquer produto de software Python compatível com um único comando. Ele também permite que você adicione esse recurso de instalação de rede ao seu próprio software Python com muito pouco trabalho.

O Python 2.7.9 e posterior (na série python2) e o Python 3.4 e posterior incluem pip por padrão.

Para ver se o pip está instalado, abra um prompt de comando e execute

```
$ command -v pip
```

Para instalar o pip, siga o guia oficial de instalação do pip - isso instalará automaticamente a versão mais recente do setuptools.

1.8.2 Ambientes virtuais

Um Virtual Environment (Ambiente virtual) é uma ferramenta que permite guardar as dependências de projetos diferentes em lugares separados criando um ambiente virtual Python para cada um deles. Isso resolve problemas como "O projeto X usa uma biblioteca na versão 1.x mas o projeto Y usa essa mesma biblioteca mas na versão 4.x" e mantém os seus pacotes instalados na pasta site-packages global limpa e organizada.

Por exemplo, você pode trabalhar em um projeto que usa o Django na versão 1.10 enquanto também poderá trabalhar em um outro projeto que use o Django mas na versão 1.8.

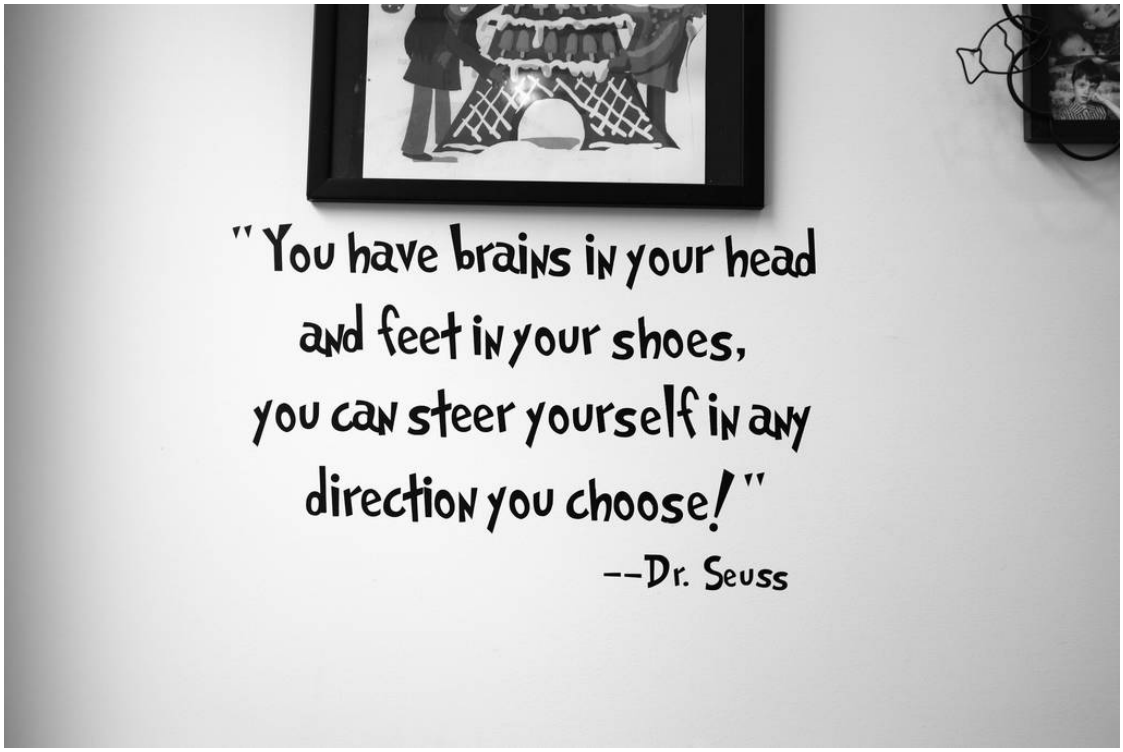
Para começar a usar isso e ver mais informações: documentação *Ambientes Virtuais*.

Você também pode usar *virtualenvwrapper* para facilitar o gerenciamento de seus ambientes virtuais.

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

- Using Virtualenvs with Pipenv:

1.9 Pipenv & Virtual Environments



This tutorial walks you through installing and using Python packages.

It will show you how to install and use the necessary tools and make strong recommendations on best practices. Keep in mind that Python is used for a great many different purposes, and precisely how you want to manage your dependencies may change based on how you decide to publish your software. The guidance presented here is most directly applicable to the development and deployment of network services (including web applications), but is also very well suited to managing development and testing environments for any kind of project.

Nota: This guide is written for Python 3, however, these instructions should work fine on Python 2.7—if you are still using it, for some reason.

1.9.1 Make sure you've got Python & pip

Before you go any further, make sure you have Python and that it's available from your command line. You can check this by simply running:

```
$ python --version
```

You should get some output like `3.6.2`. If you do not have Python, please install the latest 3.x version from python.org or refer to the Installing Python section of this guide.

Nota: If you're newcomer and you get an error like this:

```
>>> python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python' is not defined
```

It's because this command is intended to be run in a *shell* (also called a *terminal* or *console*). See the Python for Beginners getting started tutorial for an introduction to using your operating system's shell and interacting with Python.

Additionally, you'll need to make sure you have pip available. You can check this by running:

```
$ pip --version
```

If you installed Python from source, with an installer from python.org, or via Homebrew you should already have pip. If you're on Linux and installed using your OS package manager, you may have to install pip separately.

1.9.2 Installing Pipenv

Pipenv is a dependency manager for Python projects. If you're familiar with Node.js' npm or Ruby's bundler, it is similar in spirit to those tools. While pip can install Python packages, Pipenv is recommended as it's a higher-level tool that simplifies dependency management for common use cases.

Use `pip` to install Pipenv:

```
$ pip install --user pipenv
```

Nota: This does a user installation to prevent breaking any system-wide packages. If `pipenv` isn't available in your shell after installation, you'll need to add the user base's binary directory to your `PATH`.

On Linux and macOS you can find the user base binary directory by running `python -m site --user-base` and adding `bin` to the end. For example, this will typically print `~/.local` (with `~` expanded to the absolute path to your home directory) so you'll need to add `~/.local/bin` to your `PATH`. You can set your `PATH` permanently by modifying ~/.profile.

On Windows you can find the user base binary directory by running `py -m site --user-site` and replacing `site-packages` with `Scripts`. For example, this could return `C:\Users\Username\AppData\Roaming\Python36\site-packages` so you would need to set your `PATH` to include `C:\Users\Username\AppData\Roaming\Python36\Scripts`. You can set your user `PATH` permanently in the Control Panel. You may need to log out for the `PATH` changes to take effect.

1.9.3 Installing packages for your project

Pipenv manages dependencies on a per-project basis. To install packages, change into your project's directory (or just an empty directory for this tutorial) and run:

```
$ cd project_folder
$ pipenv install requests
```

Pipenv will install the excellent Requests library and create a `Pipfile` for you in your project's directory. The Pipfile is used to track which dependencies your project needs in case you need to re-install them, such as when you share your project with others. You should get output similar to this (although the exact paths shown will vary):

```
Creating a Pipfile for this project...
Creating a virtualenv for this project...
Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.
↪framework/Versions/3.6'
New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.
↪6
Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/
↪python
Installing setuptools, pip, wheel...done.

Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
Installing requests...
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Installing collected packages: idna, urllib3, chardet, certifi, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.
↪18.4 urllib3-1.22
```

(continues on next page)

(continuação da página anterior)

```
Adding requests to Pipfile's [packages]...
P.S. You have excellent taste!
```

1.9.4 Using installed packages

Now that Requests is installed you can create a simple `main.py` file to use it:

```
import requests

response = requests.get('https://httpbin.org/ip')

print('Your IP is {0}'.format(response.json()['origin']))
```

Then you can run this script using `pipenv run`:

```
$ pipenv run python main.py
```

You should get output similar to this:

```
Your IP is 8.8.8.8
```

Using `$ pipenv run` ensures that your installed packages are available to your script. It's also possible to spawn a new shell that ensures all commands have access to your installed packages with `$ pipenv shell`.

1.9.5 Next steps

Congratulations, you now know how to install and use Python packages!

1.10 Lower level: 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.

It can be used standalone, in place of Pipenv.

Instalação do virtualenv via pip:

```
$ pip install virtualenv
```

Test your installation:

```
$ virtualenv --version
```

1.10.1 Uso

1. Criar um ambiente virtual para um projeto:

```
$ cd 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.

Nota: 'venv' is the general convention used globally. As it is readily available in ignore files (eg: .gitignore')

This creates a copy of Python in whichever directory you ran the command in, placing it in a folder named `venv`.

Voc também poderá usar qualquer versão do interpretador Python se preferir (como `python2.7`).

```
$ virtualenv -p /usr/bin/python2.7 venv
```

ou definir o interpretador global com uma variável de ambiente no arquivo `~/.bashrc`:

```
$ export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python2.7
```

2. Para começar a usar o ambiente virtual, ele precisa primeiro ser ativado:

```
$ 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:project_folder 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.

For Windows, the same command mentioned in step 1 can be used to create a virtual environment. However, activating the environment requires a slightly different command.

Assuming that you are in your project directory:

```
C:\Users\SomeUser\project_folder> venv\Scripts\activate
```

Install packages using the `pip` command:

```
$ pip install requests
```

3. Se você já tiver terminado de trabalhar no ambiente virtual no momento, você pode desativá-lo:

```
$ deactivate
```

Isso faz com que você volte a versão padrão do interpretador Python do sistema e todas as suas bibliotecas.

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 it's possible you'll forget their names or where they were placed.

Nota: Python has included venv module from version 3.3. For more details: venv.

1.10.2 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 (see *Version Control Ignores*).

1.10.3 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

Uso

1. Create a virtual environment:

```
$ mkvirtualenv project_folder
```

This creates the `project_folder` folder inside `~/Envs`.

2. Work on a virtual environment:

```
$ workon project_folder
```

Alternatively, you can make a project, which creates the virtual environment, and also a project directory inside `$WORKON_HOME`, which is `cd`-ed into when you `workon project_folder`.

```
$ mkproject project_folder
```

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.

1.10.4 virtualenv-burrito

With virtualenv-burrito, you can have a working virtualenv + virtualenvwrapper environment in a single command.

1.10.5 direnv

When you `cd` into a directory containing a `.env`, direnv automagically activates the environment.

Install it on Mac OS X using `brew`:

```
$ brew install direnv
```

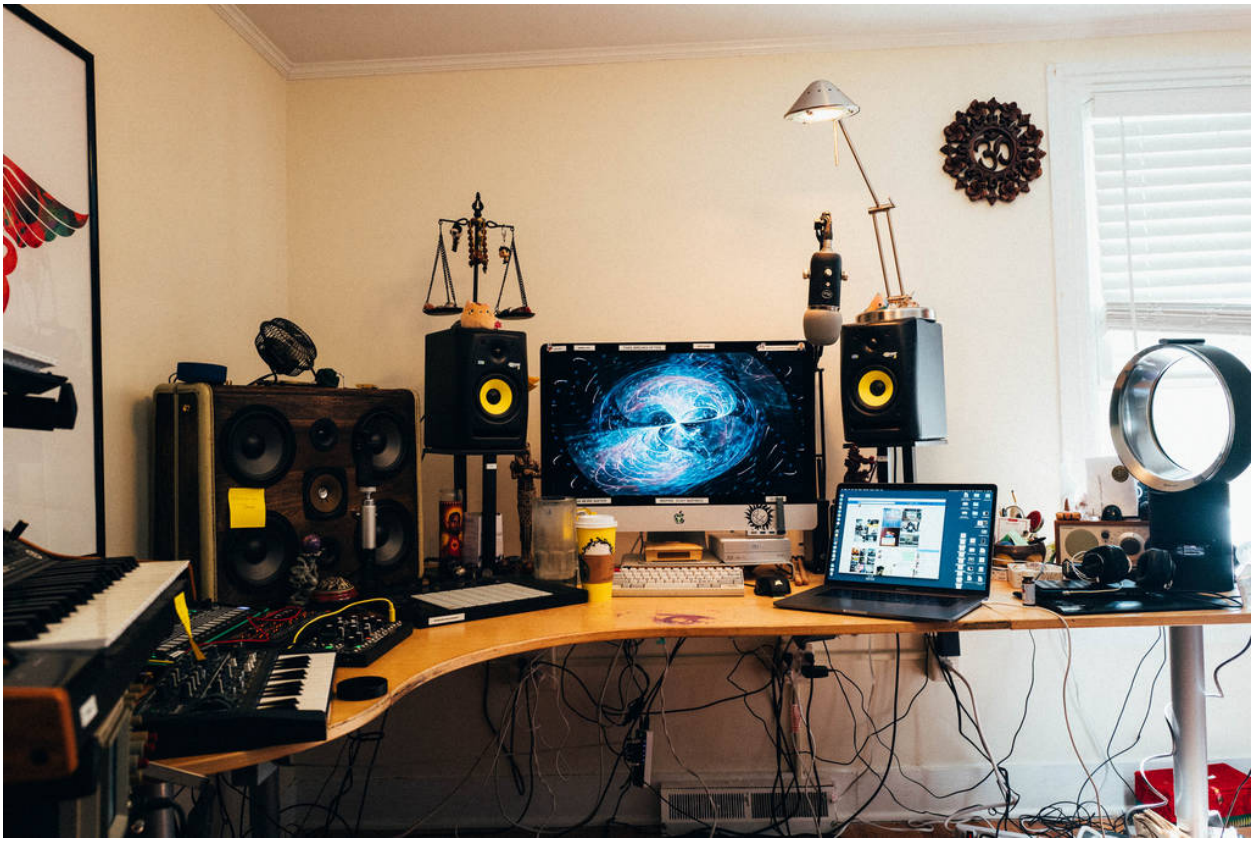
On Linux follow the instructions at direnv.net

CAPÍTULO 2

Ambientes de desenvolvimento em Python

This part of the guide focuses on the Python development environment, and the best-practice tools that are available for writing Python code.

2.1 Seu ambiente de desenvolvimento



2.1.1 Editores de texto

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

O Vim é um editor de textos que utiliza atalhos do teclado para a edição, ao invés de menus ou ícones. Existem alguns plugins e configurações para o Vim que auxiliam o desenvolvimento em Python. Se você desenvolve apenas em Python, um bom começo é definir as configurações padrão para indentação e quebras de linha para valores compatíveis com a **PEP 8**. Em seu diretório inicial, abra um arquivo chamado `.vimrc` e adicione as seguintes linhas:

```
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
```

Com essas configurações, novas linhas são inseridas após 79 caracteres e a indentação é programada para 4 espaços por pressionamento da tecla tab. Se você também usa Vim para outra línguas, existe um ótimo plugin chamado indent, que cuida da configuração da indentação para arquivos de origem do Python.

Também há um plugin de sintaxe muito prático chamado syntax, que apresenta algumas evoluções quando comparado ao arquivo de sintaxe incluído no 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 pycodestyle 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.

Para checagens de PEP8 e pyflakes, você pode instalar o vim-flake8. Agora você pode mapear a função `Flake8` para qualquer tecla de atalho ou ação que desejar no Vim. O plugin vai mostrar os erros no lado inferior da tela, e fornecer uma maneira simples de saltar para a linha correspondente. É muito útil chamar essa função sempre que você salvar um arquivo. Para fazer isso, adicione a seguinte linha ao seu `.vimrc`:

```
autocmd BufWritePost *.py call Flake8()
```

Se você já utiliza syntasic, você pode configurar ele para rodar Pyflakes na escrita e mostrar erros e advertências na janela de consertos rápidos. Um exemplo de configuração que faça isso e que também mostre o status e mensagens de advertência na barra de status pode ser:

```
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 é uma solução complexa para trabalhar com código em Python no Vim. Ele tem:

- Asynchronous Python code checking (`pylint`, `pyflakes`, `pycodestyle`, `mccabe`) in any combination
- Refatoração de código e auto-completar com Rope
- Dobra rápida Python
- Suporte virtualenv
- Busca através da documentação Python e execução de código Python
- Auto pycodestyle error fixes

E mais.

SuperTab

SuperTab é um pequeno plugin de Vim que faz a função de auto-completar mais conveniente utilizando-se a tecla `<Tab>` ou qualquer outra tecla customizada.

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. O próprio Emacs vem com o modo Python

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.

O Sublime Text possuí um suporte excelente para edição de código Python além de usar Python em sua API de plugins. Ele também possuí uma grande variedade de plugins, dos quais habilitam checagem de PEP8 no editor e "linting" de código.

Atom

> Atom é um editor de texto para o século XXI, altamente extensível, construído sobre o atom-shell e baseado em tudo o que amamos nos nossos editores favoritos.

Atom is web native (HTML, CSS, JS), focusing on modular design and easy plugin development. It comes with native package control and a plethora of packages. Recommended for Python development is Linter combined with linter-flake8.

Python (no Visual Studio Code)

Python for Visual Studio is an extension for the Visual Studio Code. This is a free, lightweight, open source code editor, 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.

licenciado MIT.

2.1.2 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.

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.

IDE Komodo

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 oferece autocompletar, destacamento de sintaxe, browser nativo, debugger gráfico e suporte para sistemas de controle de versão.

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 source code 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.

Mu

Mu is a minimalist Python IDE which can run Python 3 code locally and can also deploy code to the BBC micro:bit and to Adafruit boards running CircuitPython.

Intended for beginners, mu includes a Python 3 interpreter, and is easy to install on Windows, OS/X and Linux. It runs well on the Raspberry Pi.

There's an active support community on gitter.

2.1.3 Ferramentas do intérprete

Ambientes virtuais

Ambientes virtuais fornecem um meio poderoso para isolar dependências de pacotes do seu projeto. Isso significa que você pode usar pacotes particulares para um projeto em Python sem instalar eles em todo o seu sistema, evitando potenciais conflitos de versão.

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, while 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.

2.1.4 Outras ferramentas

IDLE

IDLE is an integrated development environment that is part of the Python standard distribution. 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.

Ele provê as seguintes funcionalidades:

- Janela terminar do Python (intérprete)
- Editor de texto com múltiplas janelas que coloriza código em Python
- Instalação de depuração mínima

IPython

O IPython oferece um kit de ferramentas para ajudar a você usar o Python interativamente. Seus principais componentes são:

- 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 its 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:

- Realce de sintaxe
- Autocompletado
- Edição multi linha
- Emacs and Vim Modes
- 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
```

2.2 Further Configuration of pip and Virtualenv

2.2.1 Requisitando um ambiente virtual ativo para o “pip”

Agora deve estar claro que usar ambientes virtuais é um ótimo meio para deixar seu ambiente virtual limpo e manter necessidades de diferentes projetos separadas.

Quando você começa a trabalhar em muitos projetos diferentes, pode ser difícil lembrar-se de ativar o ambiente virtual certo quando você volta para um projeto específico. Como resultado disso, é muito fácil instalar pacotes globalmente enquanto acha que está instalando apenas o pacote do ambiente virtual do projeto. Com o tempo isso pode ocasionar uma bagunça na sua lista de pacotes globais.

Para ter certeza que você está instalando pacotes em seu ambiente virtual ao usar “pip install”, adicione a seguinte linha ao seu `~/.bashrc` file:

```
export PIP_REQUIRE_VIRTUALENV=true
```

Após salvar essas mudanças e **sourcing** o arquivo `~/.bashrc` com `source ~/.bashrc`, o pip não vai mais deixar você instalar pacotes se você não estiver em um ambiente virtual. Se você tentar usar `pip install` fora de um ambiente virtual, o pip vai lhe lembrar que um ambiente virtual ativado é necessário para a instalação de pacotes.

```
$ pip install requests
Could not find an activated virtualenv (required).
```

Você também pode fazer essa modificação ao editar o seus arquivos `pip.conf` ou `pip.ini`. `pip.conf` é usado pelos sistemas operacionais Unix e MAC OS X e pode ser achado em:

```
$HOME/.pip/pip.conf
```

Do mesmo modo, o arquivo `pip.ini` é utilizado pelos sistemas operacionais Windows e pode ser encontrado em:

```
%USERPROFILE%\pip\pip.ini
```

Se você não possuir um arquivo `pip.conf` ou `pip.ini` nesses locais, você pode criar um novo arquivo com os nomes corretos do seu sistema operacional.

Caso você já tenha um arquivo de configuração, basta adicionar as seguintes linhas abaixo das configurações `[globais]` para requerir um ambiente virtual ativo.

```
require-virtualenv = true
```

Se você não tinha um arquivo de configuração, você terá que criar um novo e adicionar as seguintes linhas ao mesmo:

```
[global]
require-virtualenv = true
```

Você obviamente precisará instalar alguns pacotes globalmente (geralmente os que você usa em pacotes diferentes com frequência), e isso pode ser feito ao adicionar a seguinte linha ao seu arquivo `~/.bashrc`.

```
gpip() {
    PIP_REQUIRE_VIRTUALENV=false pip "$@"
}
```

Depois de salvar as alterações e recarregar o seu arquivo `~/.bashrc` você pode instalar pacotes globalmente executando `gpip install`. Você pode alterar o nome da função para qualquer coisa que você goste, mas tenha em mente que precisará usar esse nome ao tentar instalar pacotes globalmente com o `pip`.

2.2.2 Pacotes de cache para uso futuro

Cada desenvolvedor tem bibliotecas preferidas e quando você está trabalhando em um monte de projetos diferentes, é obrigado a ter alguma sobreposição entre as bibliotecas que você usa. Por exemplo, você pode estar usando a biblioteca `requests` em vários projetos diferentes.

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, starting with version 6.0, pip provides an on-by-default caching mechanism that doesn't need any configuration.

When using older versions, you can configure pip in such a way that it tries to reuse already installed packages, too.

On Unix systems, you can add the following line to your `.bashrc` or `.bash_profile` file.

```
export PIP_DOWNLOAD_CACHE=$HOME/.pip/cache
```

Você pode definir o caminho para qualquer lugar que goste (contanto que você tenha acesso de escrita). Depois de adicionar esta linha, os seus `source .bashrc` (ou `.bash_profile`) estarão todos definidos.

Outra maneira de fazer a mesma configuração é através dos arquivos `pip.conf` ou `pip.ini`, dependendo do seu sistema. Se estivermos no Windows, podemos adicionar a seguinte linha ao nosso arquivo `pip.ini` na seção `[global]`:

```
download-cache = %USERPROFILE%\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
```

Mesmo que você possa usar qualquer caminho que você goste de armazenar o seu cache, recomenda-se que você crie uma nova pasta *dentro de* onde o arquivo `pip.conf` ou o `pip.ini` está. Se você não confia em si mesmo com todo esse caminho voodoo, basta usar os valores fornecidos aqui e estará bem.

CAPÍTULO 3

Escrevendo Ótimos códigos em Python

Essa parte do guia foca-se nas melhores praticas para escrita de códigos em Python.

3.1 Estruturando seu projeto

Por “estrutura”, queremos dizer as decisões que você toma sobre a forma como o seu projeto atende melhor ao seu objetivo. Precisamos considerar como melhor aproveitar os recursos do Python para criar um código limpo e efetivo. Em termos práticos, “estrutura” significa fazer um código limpo cuja lógica e dependências são claras, bem como a forma como os arquivos e as pastas estão organizados no sistema de arquivos.

Quais funções devem entrar em quais módulos? Como os dados fluem pelo projeto? Quais recursos e funções podem ser agrupados e isolados? Ao responder perguntas como essas, você pode começar a planejar, em um sentido amplo, como será seu produto final.

In this section, we take a closer look at Python’s modules 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.

3.1.1 Estrutura do Repositório

É importante.

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.

Quando um usuário ou colaborador potencial chega à página do seu repositório, ele vê algumas coisas:

- Nome do Projeto
- Descrição do Projeto
- 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 is important.

Repositório de Amostra

tl;dr: This is what Kenneth Reitz recommended in 2013.

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
```

Vamos entrar em alguns detalhes.

The Actual Module

Localização	`./sample/` or `./sample.py`
Propósito	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.

Licença

Localização	`./LICENSE`
Propósito	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 or contributing to your code.

Setup.py

Localização	`./setup.py`
Propósito	Package and distribution management.

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

Arquivo de Requisitos

Localização	`./requirements.txt`
Propósito	Dependências de desenvolvimento.

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 if you prefer setting up a development environment via `setup.py`, this file may be unnecessary.

Documentação

Localização	`./docs/`
Propósito	Package reference documentation.

There is little reason for this to exist elsewhere.

Test Suite

For advice on writing your tests, see Testando seu código.

Localização	`./test_sample.py` ou `./tests`
Propósito	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(os.path.join(os.path.dirname(__file__), '..')))

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

Localização	`./Makefile`
Propósito	Generic management tasks.

If you look at most of my projects or any Pocoo project, you'll notice a Makefile lying around. Why? These projects aren't written in C... In short, make is an 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
```

3.1.2 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 the 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 your 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 to 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 in order to understand why a rectangular table became a square, and discover that remote template code is also modifying this context, messing with the 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) makes it very hard to maintain this kind of code. 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, then you might be swimming in ravioli code.

3.1.3 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 (?). A file name like `my.spam.py` is the 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 like, you could name your module `my_spam.py`, but even our trusty friend the underscore, should not be seen that often in module names. However, using other characters (spaces or hyphens) in module names will prevent importing (- is the subtract operator). Try to keep module names short so there is no need to separate words. And, most of all, don't namespace with underscores; use submodules instead.

```
# OK
import library.plugin.foo
# not OK
import library.foo_plugin
```

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 when it is not found.

When `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 the 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 local namespace. While much less harmful than `import *` because it shows explicitly what is imported in the local 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 *Estilo de código* 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.

3.1.4 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 `__init__.py` file in `pack` and 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 adding 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 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`.

3.1.5 Object-oriented programming

Python is sometimes described as an object-oriented programming language. This can be somewhat misleading and requires further clarifications.

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 can be considered as 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 languages.

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 as 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 some state and some functionality together. 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 as a response to external requests that happen simultaneously. In this case, holding some state in 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, the deletion may actually occur after the first process loaded the item, and then we have to mark a deleted object as read.

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) allows 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.

3.1.6 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 unrelated 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.

3.1.7 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.

3.1.8 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 the value 'a string', 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.

Mau

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

Bom

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

Using short functions or methods helps to 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:

Mau

```
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.

3.1.9 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 has 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, appending each part to the string is inefficient because the entirety of the string is copied on each append. Instead, 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. List comprehensions are usually the fastest and most idiomatic way to do this.

Mau

```
# 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)
```

Better

```
# 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'])
```

Nota: 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
```

3.1.10 Vendorizing Dependencies

3.1.11 Runners

3.1.12 Further Reading

- http://docs.python.org/3/library/
- https://diveintopython3.net/

3.2 Estilo de código

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.

3.2.1 Conceitos gerais

Código Explicito

Embora seja possível qualquer tipo de magia negra com o Python, a maneira mais explícita e direta é a preferida.

Mau

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

Bom

```
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.

Uma declaração por linha

Embora algumas declarações compostas, como compreensões de lista, sejam permitidas e apreciadas pela sua brevidade e sua expressividade, é má prática ter duas declarações desconexas na mesma linha de código.

Mau

```
print('one'); print('two')

if x == 1: print('one')

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

Bom

```
print('one')
print('two')

if x == 1:
    print('one')

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

Argumentos de Funções

Argumentos podem ser passados para funções de quatro maneiras diferentes.

1. **Os argumentos posicionais** são obrigatórios e não têm valores padrão. Eles são a forma mais simples de argumentos e podem ser usados para os poucos argumentos de função que fazem parte integrante do significado da função e sua ordem é natural. Por exemplo, no `send (message, recipient)` ou `point (x, y)` o

usuário da função não tem dificuldade em lembrar que essas duas funções requerem dois argumentos e em qual ordem.

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 the 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 others', 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.
```

3.2.2 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('.')
```

Nota: 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 range(4)]
```

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.

3.2.3 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.

3.2.4 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, pycodestyle (previously known as `pep8`), that can check your code for conformance. Install it by running the following command in your terminal:

```
$ pip install pycodestyle
```

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

```
$ pycodestyle 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'
```

Auto-Formatting

There are several auto-formatting tools that can reformat your code, in order to comply with PEP 8.

autopep8

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.

yapf

While autopep8 focuses on solving the PEP 8 violations, yapf tries to improve the format of your code aside from complying with PEP 8. This formatter aims at providing as good looking code as a programmer who writes PEP 8 compliant code. It gets installed with:

```
$ pip install yapf
```

Run the auto-formatting of a file with:

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

Similar to autopep8, running the command without the `--in-place` flag will output the diff for review before applying the changes.

black

The auto-formatter black offers an opinionated and deterministic reformatting of your code base. Its main focus lies in providing a uniform code style without the need of configuration throughout its users. Hence, users of black are able to forget about formatting altogether. Also, due to the deterministic approach minimal git diffs with only the relevant changes are guaranteed. You can install the tool as follows:

```
$ pip install black
```

A python file can be formatted with:

```
$ black optparse.py
```

Adding the `--diff` flag provides the code modification for review without direct application.

3.2.5 Conventions

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

Check if a 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 provides a powerful, concise way to work with lists.

Generator expressions follows almost the same syntax as list comprehensions but return a generator instead of a list.

Creating a new list requires more work and uses more memory. If you are just going to loop through the new list, prefer using an iterator instead.

Bad:

```
# needlessly allocates a list of all (gpa, name) entires in memory
valedictorian = max([(student.gpa, student.name) for student in graduates])
```

Good:

```
valedictorian = max((student.gpa, student.name) for student in graduates)
```

Use list comprehensions when you really need to create a second list, for example if you need to use the result multiple times.

If your logic is too complicated for a short list comprehension or generator expression, consider using a generator function instead of returning a list.

Good:

```
def make_batches(items, batch_size):
    """
    >>> list(make_batches([1, 2, 3, 4, 5], batch_size=3))
    [[1, 2, 3], [4, 5]]
    """
    current_batch = []
    for item in items:
        current_batch.append(item)
        if len(current_batch) == batch_size:
            yield current_batch
            current_batch = []
    yield current_batch
```

Never use a list comprehension just for its side effects.

Bad:

```
[print(x) for x in sequence]
```

Good:

```
for x in sequence:
    print(x)
```

Filtering a list

Bad:

Never remove items from a list while you are iterating through it.

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

Don't make multiple passes through the list.

```
while i in a:
    a.remove(i)
```

Good:

Use a list comprehension or generator expression.

```
# comprehensions create a new list object
filtered_values = [value for value in sequence if value != x]

# generators don't create another list
filtered_values = (value for value in sequence if value != x)
```

Possible side effects of modifying the original list

Modifying the original list can be risky if there are other variables referencing it. But you can use *slice assignment* if you really want to do that.

```
# replace the contents of the original list
sequence[::] = [value for value in sequence if value != x]
```

Modifying the values in a list

Bad:

Remember that assignment never creates a new object. If two or more variables refer to the same list, changing one of them changes them all.

```
# Add three to all list members.
a = [3, 4, 5]
b = a                       # a and b refer to the same list object

for i in range(len(a)):
    a[i] += 3               # b[i] also changes
```

Good:

It's safer to create a new list object and leave the original alone.

```
a = [3, 4, 5]
b = a

# assign the variable "a" to a new list without changing "b"
a = [i + 3 for i in 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, "
```

(continues on next page)

(continuação da página anterior)

```
    "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.

3.3 Lendo Ótimos Códigos

Um dos segredos para tornar-se um excelente programador Python é lendo, entendendo e compreendendo códigos excelentes.

O código excelente geralmente segue as diretrizes descritas em *Estilo de código*, e faz o melhor para expressar uma intenção clara e concisa para o leitor.

Incluído abaixo temos uma lista de projetos Python recomendados para leitura. Cada um desses projetos é um paradigma de codificação Python.

- Howdoi Howdoi é uma ferramenta de busca de código, escrito em Python.
- Flask O Flask é um microframework para Python baseado no Werkzeug e Jinja2. O objetivo é obter um avanço inicial rápido e foi desenvolvido tendo as melhores ideias e intenções.

- 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 a 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 `request` é uma biblioteca HTTP habilitada para Apache2, escrita em Python, e desenvolvido para seres humanos.
- Tablib O `tablib` é uma biblioteca de conjunto de dados tabulares, agnóstica de formato, escrita em Python.

Por fazer: Incluir exemplos de código demonstrativos de cada um dos projetos listados. Explicar por que o mesmo é um código excelente. Use exemplos complexos.

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

3.4 Documentação

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.

3.4.1 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.

3.4.2 Project Publication

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

- An *introduction* should give 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.

Nota: 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, but with all the optional extensions built in.

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

3.4.3 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. 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 optimized 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 towards explaining 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 used, is the one used by the NumPy project, often called NumPy style docstrings. While it can take up more lines than 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**

3.4.4 Outras ferramentas

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.

3.5 Testando seu código

Testar o seu código é muito importante.

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

Algumas regras gerais de teste:

- Uma unidade de teste deve se concentrar em um pequeno número de funcionalidades e provar que tudo está correto.
- Cada unidade de teste deve ser totalmente independente. Cada teste deve ser capaz de executar sozinho, e também dentro do conjunto de teste, independentemente da ordem em que são chamados. A implicação desta regra é que cada teste deve ser carregado com um novo conjunto de dados e talvez seja necessário fazer alguma limpeza depois. Isso geralmente é gerenciado pelos métodos `setUp()` e `tearDown()`.
- Tente arduamente fazer testes que funcionem rapidamente. Se um único teste precisa de mais de alguns milissegundos para executar, o desenvolvimento será diminuído ou os testes não serão executados com a frequência desejável. Em alguns casos, os testes não podem ser rápidos porque eles precisam de uma estrutura de dados complexa para trabalhar, e esta estrutura de dados deve ser carregada toda vez que o teste é executado. Mantenha esses testes mais pesados em um conjunto de teste separado executado por alguma tarefa agendada e execute todos os outros testes sempre que necessário.
- Aprenda suas ferramentas e aprenda como executar uma única prova ou um caso de teste. Então, ao desenvolver uma função dentro de um módulo, execute os testes dessa função freqüentemente, idealmente automaticamente quando você salvar o código.
- Execute sempre o conjunto de teste completo antes de uma sessão de codificação, e execute novamente depois. Isso lhe dará mais confiança de que você não quebrou nada no resto do código.
- É uma boa ideia implementar um hook que executa todos os testes antes de enviar o código para um repositório compartilhado.
- Caso estejas no meio de uma sessão de desenvolvimento e tiver que interromper o seu trabalho, é uma boa ideia escrever um teste de unidade quebrado sobre o que deseja desenvolver em seguida. Ao retornar ao desenvolvimento, você terá um ponteiro para onde estavas e voltará ao percurso mais rapidamente.
- O primeiro passo quando você estiver depurando o seu código é escrever um novo teste identificando o bug. Embora nem sempre seja possível, esses testes de detecção de falhas estão entre os mais valiosos itens de código do seu projeto.
- Use nomes longos e descritivos para testar funções. O guia de estilo aqui é ligeiramente diferente do código de execução, onde os nomes curtos são frequentemente preferidos. O motivo é que as funções de teste nunca são chamadas explicitamente. `square()` ou mesmo `sqr()` está ok no código em execução, mas no código de teste você teria nomes como `test_square_of_number_2()`, `test_square_negative_number()`. Esses nomes de função são exibidos quando um teste falhar e deverá ser o mais descritivo possível.
- 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.

3.5.1 O Básico

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.

Criar casos de teste será realizado escrevendo uma subclasse `unittest.TestCase`.

```
import unittest

def fun(x):
    return x + 1

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

A partir do Python 2.7 o unittest também inclui seus próprios mecanismos de descoberta de teste.

> 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.

3.5.2 Ferramentas

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

Hypothesis

Hypothesis is a library which lets you write tests that are parameterized by a source of examples. It then generates simple and comprehensible examples that make your tests fail, letting you find more bugs with less work.

```
$ pip install hypothesis
```

For example, testing lists of floats will try many examples, but report the minimal example of each bug (distinguished exception type and location):

```
@given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
def test_mean(xs):
    mean = sum(xs) / len(xs)
    assert min(xs) <= mean(xs) <= max(xs)
```

```
Falsifying example: test_mean(
    xs=[1.7976321109618856e+308, 6.102390043022755e+303]
)
```

Hypothesis is practical as well as very powerful and will often find bugs that escaped all other forms of testing. It integrates well with py.test, and has a strong focus on usability in both simple and advanced scenarios.

hypothesis

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

mock

`unittest.mock` é uma biblioteca para a realização de testes em Python. A partir da versão do Python 3.3, está disponível a standard library.

Para as versões mais antigas do 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 with which you can configure and control its behaviour.

mock

3.6 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.

As an alternative, loguru provides an approach for logging, nearly as simple as using a simple `print` statement.

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.

3.6.1 ... 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`.

3.6.2 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:

Nota: 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 the best practice from the requests source – place this in your `__init__.py`:

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

3.6.3 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 the source code.

Example Configuration via an INI File

Let us say that 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
```

(continues on next page)

(continuação da página anterior)

```
[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')
```

3.7 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 for 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 are generally sensible, once you're aware of the underlying cause for the surprise.

3.7.1 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 Actually Happens

```
[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
```

Do not forget, you are passing a *list* object as the second argument.

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.

3.7.2 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 Actually Happens

```
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.

3.7.3 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, 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.

Version Control Ignores

If you still need the `.pyc` files for performance reasons, you can always add them to the ignore files of your version control repositories. Popular version control systems have the ability to use wildcards defined in a file to apply special rules.

An ignore file will make sure the matching files don't get checked into the repository. Git uses `.gitignore` while Mercurial uses `.hgignore`.

At the minimum your ignore files should look like this.

```
syntax:glob   # This line is not needed for .gitignore files.
*.py[cod]     # Will match .pyc, .pyo and .pyd files.
__pycache__/  # Exclude the whole folder
```

You may wish to include more files and directories depending on your needs. The next time you commit to the repository, these files will not be included.

3.8 Escolhendo uma licença

Your source publication *needs* a license. In the US, unless a license is specified, users have no legal right to download, modify, or distribute the product. 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, and 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.

CAPÍTULO 4

Guia de cenário para aplicações em Pyhton

Essa parte do guia foca-se em dicas de ferramentas e módulos baseados em diferentes cenários.

4.1 Aplicações de rede

4.1.1 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

4.1.2 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

4.2 Aplicações web & Frameworks

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

4.2.1 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**.

4.2.2 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 Encapsulates 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, Europe, and Australia.

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.

Falcon

Falcon is a good choice when your goal is to build RESTful API microservices that are fast and scalable.

It is a reliable, high-performance Python web framework for building large-scale app backends and microservices. Falcon encourages the REST architectural style of mapping URIs to resources, trying to do as little as possible while remaining highly effective.

Falcon highlights four main focuses: speed, reliability, flexibility, and debuggability. It implements HTTP through "responders" such as `on_get()`, `on_put()`, etc. These responders receive intuitive request and response objects.

Tornado

Tornado is an asynchronous 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. A set of provided cookiecutter templates helps making new project decisions for users. It powers one of the most important parts of python infrastructure PyPI.

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.

Masonite

Masonite is a modern and developer centric, "batteries included", web framework.

The Masonite framework follows the MVC (Model-View-Controller) architecture pattern and is heavily inspired by frameworks such as Rails and Laravel, so if you are coming to Python from a Ruby or PHP background then you will feel right at home!

Masonite comes with a lot of functionality out of the box including a powerful IOC container with auto resolving dependency injection, craft command line tools, and the Orator active record style ORM.

Masonite is perfect for beginners or experienced developers alike and works hard to be fast and easy from install through to deployment. Try it once and you'll fall in love.

FastAPI

FastAPI is a modern web framework for building APIs with Python 3.6+.

It has very high performance as it is based on Starlette and Pydantic.

FastAPI takes advantage of standard Python type declarations in function parameters to declare request parameters and bodies, perform data conversion (serialization, parsing), data validation, and automatic API documentation with **OpenAPI 3** (including **JSON Schema**).

It includes tools and utilities for security and authentication (including OAuth2 with JWT tokens), a dependency injection system, automatic generation of interactive API documentation, and other features.

4.2.3 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.

4.2.4 WSGI Servers

Stand-alone WSGI servers typically use less resources than traditional web servers and provide top performance[1].

[1] 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.

4.2.5 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.

4.2.6 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.

4.2.7 Templating

Most WSGI applications are responding to HTTP requests to serve content in HTML or other markup languages. Instead of directly generating 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 of 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 are 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
```

(continues on next page)

(continuação da página anterior)

```
# 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" %}
```

(continues on next page)

(continuação da página anterior)

```
{% 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}
```

(continues on next page)

(continuação da página anterior)

```
        </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 are 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.

Referências

4.3 HTML Scraping

4.3.1 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.

4.3.2 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.

4.4 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

4.4.1 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.

4.4.2 docopt

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

4.4.3 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 imperatively. This module targets unsophisticated users, programmers, sysadmins, 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.

4.4.4 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 `svn` and `git`, where the main program handles some basic argument parsing and then invokes a sub-command to do the work.

4.4.5 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 its use cases span from the simplicity of a micro-framework to the complexity of a mega-framework.

4.4.6 Python Fire

Python Fire is a library for automatically generating command-line interfaces from absolutely any Python object. It can help debug Python code more easily from the command line, create CLI interfaces to existing code, allow you to interactively explore code in a REPL, and simplify transitioning between Python and Bash (or any other shell).

4.5 GUI Applications

Alphabetical list of GUI Applications.

4.5.1 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.

4.5.2 Cocoa

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

4.5.3 GTk

Nota: PyGTK provides Python bindings for the GTK+ toolkit. However, it has been superseded by PyGObject. PyGTK should not be used for new projects and existing projects should be ported to PyGObject.

4.5.4 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

4.5.5 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, OS X, Windows, Android).

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

4.5.6 PyObjC

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

4.5.7 PySide

PySide is a Python binding of the cross-platform GUI toolkit Qt. The package name depends on the major Qt version (*PySide* for Qt4, *PySide2* for Qt5, and *PySide6* for Qt6). This set of bindings is developed by The Qt Company.

```
$ pip install pyside6
```

https://pyside.org

4.5.8 PyQt

Nota: 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

4.5.9 Pyjs Desktop (formerly Pyjamas Desktop)

Pyjs Desktop is a application widget set for desktop and a cross-platform framework. It allows the exact same Python web application source code to be executed as a standalone desktop application.

The main website: pyjs.

4.5.10 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.

4.5.11 PySimpleGUI

PySimpleGUI is a wrapper for Tkinter and Qt (others on the way). The amount of code required to implement custom GUIs is much shorter using PySimpleGUI than if the same GUI were written directly using Tkinter or Qt. PySimpleGUI code can be "ported" between GUI frameworks by changing import statements.

```
$ pip install pysimplegui
```

PySimpleGUI is contained in a single PySimpleGUI.py file. Should pip installation be impossible, copying the PySimpleGUI.py file into a project's folder is all that's required to import and begin using.

4.5.12 Toga

Toga is a Python native, OS native, cross platform GUI toolkit. Toga consists of a library of base components with a shared interface to simplify platform-agnostic GUI development.

Toga is available on macOS, Windows, Linux (GTK), and mobile platforms such as Android and iOS.

4.5.13 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.

4.5.14 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 https://www.wxpython.org/pages/downloads/ and download the appropriate package for your OS.*

4.6 Bancos de dados

4.6.1 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.

Podem ser encontrados tutoriais que explicam como trabalhar com módulos que estejam em conformidade com esta interface aqui e aqui.

4.6.2 SQLAlchemy

O SQLAlchemy é um conjunto de ferramentas de banco de dados comum. Ao contrário de muitas bibliotecas de banco de dados, ele não só fornece uma camada ORM, mas também uma API generalizada para escrever o código agnóstico do banco de dados sem SQL.

```
$ pip install sqlalchemy
```

4.6.3 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
```

Também está incluída uma ferramenta de linha de comando para exportação de dados SQL.

4.6.4 PugSQL

PugSQL is a simple Python interface for organizing and using parameterized, handwritten SQL. It is an anti-ORM that is philosophically lo-fi, but it still presents a clean interface in Python.

```
$ pip install pugsql
```

4.6.5 Django ORM

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

Baseia-se na ideia de modelos, uma abstração que facilita a manipulação de dados em Python.

O básico:

- Cada modelo é uma classe Python e que é uma subclasse de `django.db.models.Model`.
- Cada atributo do modelo representa um campo de banco de dados.
- O Django oferece uma API de acesso ao banco de dados gerada automaticamente; Making queries.

4.6.6 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 PostgreSQL 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 PostgreSQL are supported out-of-the-box, there is a collection of add-ons available.

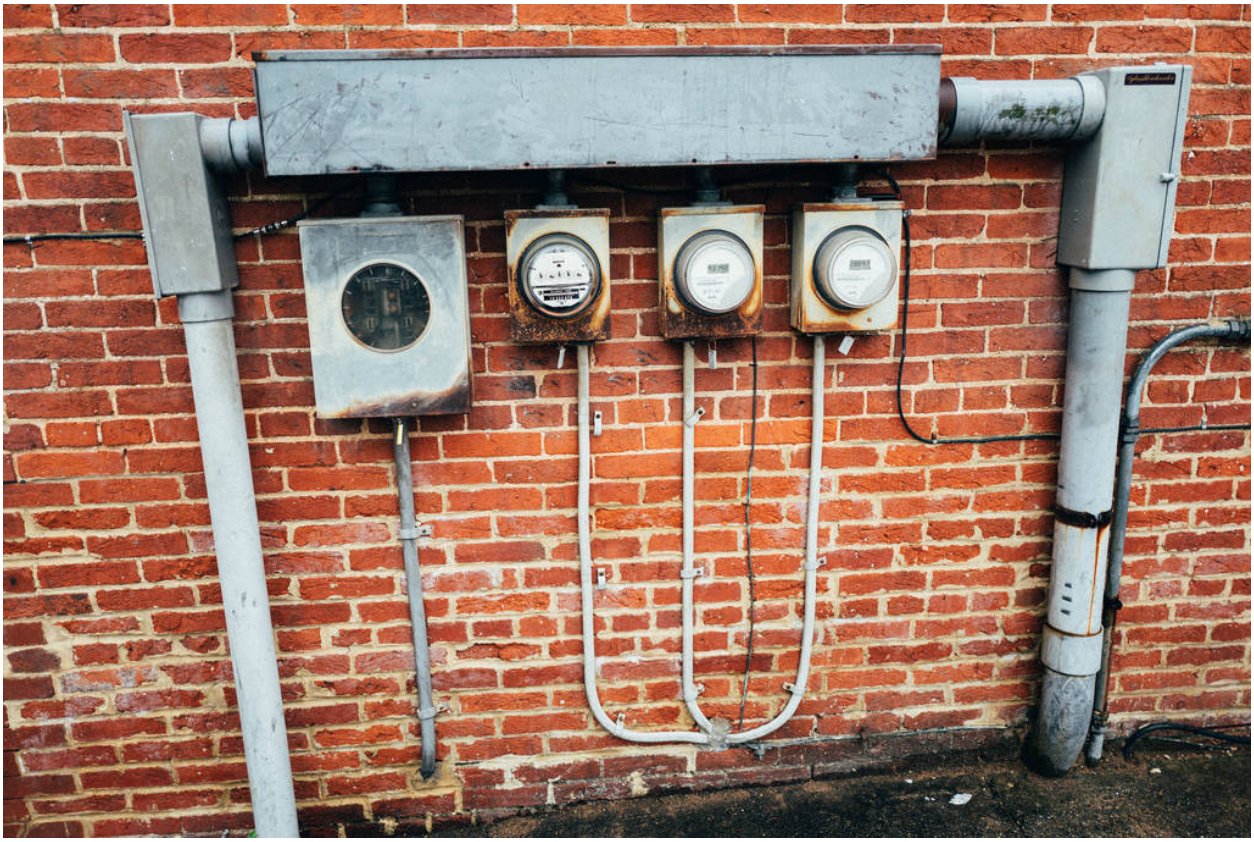
4.6.7 PonyORM

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 a 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, PostgreSQL, and Oracle.

4.6.8 SQLObject

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

4.7 Networking



4.7.1 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.

4.7.2 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: conecta um conjunto de clientes a um conjunto de serviços. Esta é uma chamada de procedimento remoto e um padrão de distribuição de tarefas.
- publish-subscribe: conecta um conjunto de editores a um conjunto de assinantes. Este é um padrão de distribuição de dados.
- 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.

Para ler um guia rápido veja ZeroMQ guide.

4.7.3 gevent

gevent é uma biblioteca de rede Python baseada em corutine que usa greenlets para fornecer uma API síncrona de alto nível em cima do loop de eventos libev.

4.8 Administração de sistemas

4.8.1 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']

```

(continues on next page)

(continuação da página anterior)

```
@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

4.8.2 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

4.8.3 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
```

(continues on next page)

(continuação da página anterior)

```
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.

4.8.4 Ansible

Ansible is an open source system automation tool. Its biggest advantage over Puppet or Chef is that 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.

4.8.5 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 DigitalOcean 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

4.8.6 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 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 manifests 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 the 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 every time 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

4.8.7 Blueprint

Por fazer: Write about Blueprint

4.8.8 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.

4.8.9 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, Linux, and FreeBSD.

4.9 Integração contínua

Nota: For advice on writing your tests, see *Testando seu código*.

4.9.1 Why?

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

4.9.2 Jenkins

Jenkins CI is an extensible Continuous Integration engine. Use it.

4.9.3 Buildbot

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

4.9.4 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

4.9.5 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.

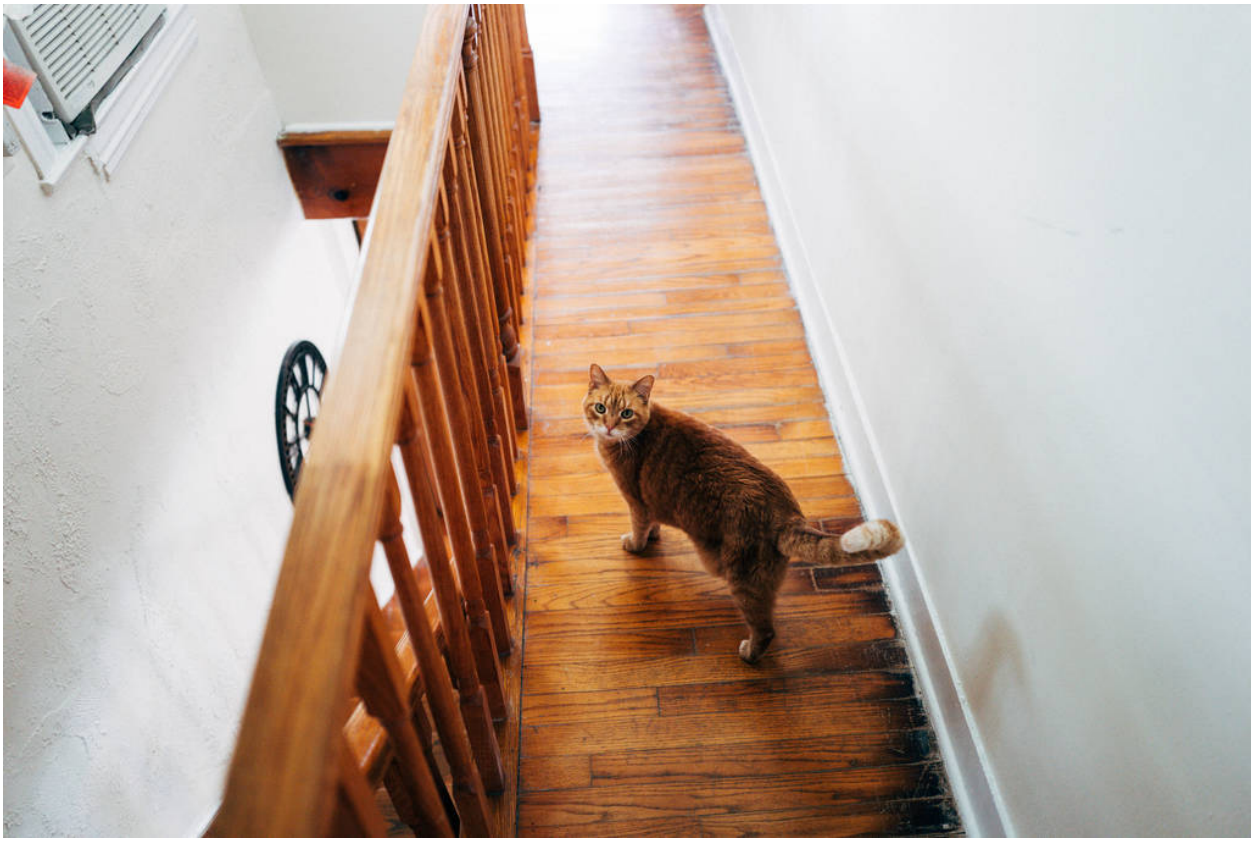
In order to get started, add a `.travis.yml` file to your repository with this example content:

```
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.

4.10 Velocidade



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
```

4.10.1 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.

4.10.2 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()
import primesCy  # primes implemented with Cython
import primes  # primes implemented with Python

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

(continues on next page)

(continuação da página anterior)

```
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 is the output of an embedded ARM beaglebone machine:

```
Cython time: 0.0196 seconds

Python time: 0.3302 seconds
```

Pyrex

Shedskin?

4.10.3 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:
```

(continues on next page)

(continuação da página anterior)

```
    # 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 its 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 Stack Overflow 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

4.11 Aplicações científicas

4.11.1 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.

4.11.2 Ferramentas

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 alongside the results and additional comments (HTML, LaTeX, Markdown). The notebook can then be shared and exported in various file formats.

4.11.3 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 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 recreate them, are available in the matplotlib gallery.

Pandas

Pandas is a 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.

xarray

xarray is similar to Pandas, but it is intended for wrapping multidimensional scientific data. By labelling the data with dimensions, coordinates, and attributes, it makes complex multidimensional operations clearer and more intuitive. It also wraps matplotlib for quick plotting, and can apply most operations in parallel using dask.

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.

4.11.4 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

The Anaconda Python Distribution includes all the common scientific Python packages as well as many packages related to data analytics and big data. Anaconda itself is free, and a number of proprietary add-ons are available for a fee. 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.

4.12 Manipulação de imagem

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

A brief description of both is given below.

4.12.1 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 major operating systems, and supports Python 3.

Instalação

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
```

Exemplo

```
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.

4.12.2 Open Source Computer Vision

Open Source 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.

Instalação

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
```

Exemplo

```
import cv2
#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.

4.13 Serialização de dados

4.13.1 O que é serialização de dados?

Data serialization is the process of converting structured data to a format that allows sharing or storage of the data in a form that allows recovery of its original structure. In some cases, the secondary intention of data serialization is to minimize the data's size which then reduces disk space or bandwidth requirements.

4.13.2 Flat vs. Nested data

Before beginning to serialize data, it is important to identify or decide how the data should be structured during data serialization - flat or nested. The differences in the two styles are shown in the below examples.

Flat style:

```
{ "Type" : "A", "field1": "value1", "field2": "value2", "field3": "value3" }
```

Nested style:

```
{"A"
    { "field1": "value1", "field2": "value2", "field3": "value3" } }
```

For more reading on the two styles, please see the discussion on Python mailing list, IETF mailing list and in stackexchange.

4.13.3 Serializing Text

Simple file (flat data)

If the data to be serialized is located in a file and contains flat data, Python offers two methods to serialize data.

repr

The repr method in Python takes a single object parameter and returns a printable representation of the input:

```
# input as flat text
a =  { "Type" : "A", "field1": "value1", "field2": "value2", "field3": "value3" }

# the same input can also be read from a file
a = open('/tmp/file.py', 'r')

# returns a printable representation of the input;
# the output can be written to a file as well
print(repr(a))

# write content to files using repr
with open('/tmp/file.py') as f:f.write(repr(a))
```

ast.literal_eval

The literal_eval method safely parses and evaluates an expression for a Python datatype. Supported data types are: strings, numbers, tuples, lists, dicts, booleans, and None.

```
with open('/tmp/file.py', 'r') as f: inp = ast.literal_eval(f.read())
```

CSV file (flat data)

The CSV module in Python implements classes to read and write tabular data in CSV format.

Simple example for reading:

```
# Reading CSV content from a file
import csv
with open('/tmp/file.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Simple example for writing:

```
# Writing CSV content to a file
import csv
with open('/temp/file.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(iterable)
```

The module's contents, functions, and examples can be found in the Python documentation.

YAML (nested data)

There are many third party modules to parse and read/write YAML file structures in Python. One such example is below.

```
# Reading YAML content from a file using the load method
import yaml
with open('/tmp/file.yaml', 'r', newline='') as f:
    try:
        print(yaml.load(f))
    except yaml.YAMLError as ymlexcp:
        print(ymlexcp)
```

Documentation on the third party module can be found in the PyYAML Documentation.

JSON file (nested data)

Python's JSON module can be used to read and write JSON files. Example code is below.

Reading:

```
# Reading JSON content from a file
import json
with open('/tmp/file.json', 'r') as f:
    data = json.load(f)
```

Writing:

```
# Writing JSON content to a file using the dump method
import json
with open('/tmp/file.json', 'w') as f:
    json.dump(data, f, sort_keys=True)
```

XML (nested data)

XML parsing in Python is possible using the *xml* package.

Example:

```
# reading XML content from a file
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

More documentation on using the *xml.dom* and *xml.sax* packages can be found in the Python XML library documentation.

4.13.4 Binary

NumPy Array (flat data)

Python's NumPy array can be used to serialize and deserialize data to and from byte representation.

Example:

```
import NumPy as np

# Converting NumPy array to byte format
byte_output = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]).tobytes()

# Converting byte format back to NumPy array
array_format = np.frombuffer(byte_output)
```

Pickle (nested data)

O módulo de serialização de dados nativos para Python é chamado Pickle.

Aqui está um exemplo:

```
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 )
```

4.13.5 Protobuf

Se você está procurando por um módulo de serialização com suporte em vários idiomas, a biblioteca Protobuf do Google é uma opção.

4.14 Análise de XML

4.14.1 untangle

untangle é uma biblioteca simples que recebe um documento XML e retorna um objeto Python contendo os nós e atributos em sua estrutura.

Por exemplo, um arquivo XML como este:

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

pode ser carregado assim:

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

and then you can get the child element's name attribute like this:

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

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

4.14.2 xmltodict

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

Um arquivo XML como este:

```
<mydocument has="an attribute">
  <and>
    <many>elements</many>
    <many>more elements</many>
  </and>
  <plus a="complex">
    element as well
  </plus>
</mydocument>
```

pode ser carregado em um dicionário Python como este:

```
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 XML namespaces.

4.15 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.

4.15.1 Analisando Arquivos JSON

Pegue a seguinte String contendo dados JSON:

```
json_string = '{"first_name": "Guido", "last_name":"Rossum"}'
```

A mesma pode ser analisado assim:

```
import json
parsed_json = json.loads(json_string)
```

e agora pode ser usado como um dicionário normal:

```
print(parsed_json['first_name'])
"Guido"
```

Também podemos pode converte-la da seguinte forma para JSON:

```
d = {
    'first_name': 'Guido',
    'second_name': 'Rossum',
```

(continues on next page)

(continuação da página anterior)

```
    'titles': ['BDFL', 'Developer'],
}

print(json.dumps(d))
'{"first_name": "Guido", "last_name": "Rossum", "titles": ["BDFL", "Developer"]}'
```

4.16 Criptografia

4.16.1 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 a simple API for proper symmetric encryption and the hazmat layer provides low-level cryptographic primitives.

Instalação

```
$ pip install cryptography
```

Exemplo

Exemplo de código usando receita de criptografia simétrica de alto nível:

```
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)
```

4.16.2 GPGME bindings

The GPGME Python bindings provide Pythonic access to GPG Made Easy, a C API for the entire GNU Privacy Guard suite of projects, including GPG, libgcrypt, and gpgsm (the S/MIME engine). It supports Python 2.6, 2.7, 3.4, and above. Depends on the SWIG C interface for Python as well as the GnuPG software and libraries.

A more comprehensive GPGME Python Bindings HOWTO is available with the source, and an HTML version is available at http://files.au.adversary.org. Python 3 sample scripts from the examples in the HOWTO are also provided with the source and are accessible at gnupg.org.

Available under the same terms as the rest of the GnuPG Project: GPLv2 and LGPLv2.1, both with the "or any later version" clause.

Instalação

Included by default when compiling GPGME if the configure script locates a supported python version (which it will if it's in $PATH during configuration).

Exemplo

```
import gpg

# Encryption to public key specified in rkey.
a_key = input("Enter the fingerprint or key ID to encrypt to: ")
filename = input("Enter the filename to encrypt: ")
with open(filename, "rb") as afile:
    text = afile.read()
c = gpg.core.Context(armor=True)
rkey = list(c.keylist(pattern=a_key, secret=False))
ciphertext, result, sign_result = c.encrypt(text, recipients=rkey,
                                            always_trust=True,
                                            add_encrypt_to=True)
with open("{0}.asc".format(filename), "wb") as bfile:
    bfile.write(ciphertext)
# Decryption with corresponding secret key
# invokes gpg-agent and pinentry.
with open("{0}.asc".format(filename), "rb") as cfile:
    plaintext, result, verify_result = gpg.Context().decrypt(cfile)
with open("new-{0}".format(filename), "wb") as dfile:
    dfile.write(plaintext)
# Matching the data.
# Also running a diff on filename and the new filename should match.
if text == plaintext:
    print("Hang on ... did you say *all* of GnuPG?  Yep.")
```

(continues on next page)

(continuação da página anterior)

```
else:
    pass
```

4.17 Aprendizado de máquina

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 listed below.

4.17.1 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.

Instalação

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.

4.17.2 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 ease of, it's one of the most widely-used libraries for many Machine Learning applications.

Instalação

Through PyPI:

```
pip install -U scikit-learn
```

Through conda:

```
conda install scikit-learn
```

scikit-learn also comes shipped with Anaconda (mentioned above). For more installation instructions, refer to this link.

Exemplo

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()
```

(continues on next page)

(continuação da página anterior)

```
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.

4.18 Interfacing with C/C++ Libraries

4.18.1 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))
```

4.18.2 ctypes

ctypes is the de facto standard 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 brings 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/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)]
```

4.18.3 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"
```

(continues on next page)

(continuação da página anterior)

```
%}

%extend MyClass {
    std::string __repr__()
    {
        return $self->getName();
    }
}

%include "MyClass.h"
```

4.18.4 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.

CAPÍTULO 5

Entregando um ótimo código em Python

This part of the guide focuses on sharing and deploying your Python code.

5.1 Publishing Your Code

Por fazer: Replace this kitten with the photo we want.

A healthy open source project needs a place to publish its code and project management stuff so other developers can collaborate with you. This lets your users gain a better understanding of your code, keep up with new developments, report bugs, and contribute code.

This development web site should include the source code history itself, a bug tracker, a patch submission (aka "Pull Request") queue, and possibly additional developer-oriented documentation.

There are several free open source project hosting sites (aka "forges"). These include GitHub, SourceForge, Bitbucket, and GitLab. GitHub is currently the best. Use GitHub.

5.1.1 Creating a Project Repo on GitHub

To publish your Python project on GitHub:

1. Create a GitHub account if you don't already have one.
2. Create a new repo for your project.
   (a) Click on the "+" menu next to your avatar in the upper right of the page and choose "New repository".
   (b) Name it after your project and give it an SEO-friendly description.
   (c) If you don't have an existing project repo, choose the settings to add a README, *.gitignore*, and license. Use the Python *.gitignore* option.
3. On the newly created repo page, click "Manage topics" and add the tags "python" and "python3" and/or "python2" as appropriate.

4. Include a link to your new GitHub repo in your project's README file so people who just have the project distribution know where to find it.

If this is a brand new repo, clone it to your local machine and start working:

```
$ git clone https://github.com/<username>/<projectname>
```

Or, if you already have a project Git repo, add your new GitHub repo as a remote:

```
$ cd <projectname>
$ git remote add origin https://github.com/<username>/<projectname>
$ git push --tags
```

5.1.2 When Your Project Grows

For more information about managing an open source software project, see the book Producing Open Source Software.

5.2 Empacotando o Seu Código

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.

5.2.1 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.)

5.2.2 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 http.server 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`.

5.2.3 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 2-12 MB 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 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 OS X. 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 OS X.

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)

5.3 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 2–12 MB. Also, you will be responsible for shipping updated versions of your application when security vulnerabilities to Python are patched.

5.3.1 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.)

Por fazer: Fill in "Freezing Your Code" stub

5.3.2 Comparison of Freezing Tools

Date of this writing: Oct 5, 2019 Solutions and platforms/features supported:

Solution	Windows	Linux	OS X	Python 3	Licença	One-file mode	Zipfile import	Eggs	pkg_resources support	Latest release date
bbFreeze	yes	yes	yes	no	MIT	no	yes	yes	yes	Jan 20, 2014
py2exe	yes	no	no	yes	MIT	yes	yes	no	no	Oct 21, 2014
PyInstaller	yes	yes	yes	yes	GPL	yes	no	yes	no	Jul 9, 2019
cx_Freeze	yes	yes	yes	yes	PSF	no	yes	yes	no	Aug 29, 2019
py2app	no	no	yes	yes	MIT	no	yes	yes	yes	Mar 25, 2019

Nota: Freezing Python code on Linux into a Windows executable was only once supported in PyInstaller and later dropped.

Nota: All solutions need a Microsoft Visual C++ to be installed on the target machine, except py2app. Only PyInstaller makes a self-executable exe that bundles the appropriate DLL when passing `--onefile` to `Configure.py`.

5.3.3 Windows

bbFreeze

Prerequisite is to install *Python, Setuptools and pywin32 dependency on Windows*.

1. Install `bbfreeze`:

```
$ pip install bbfreeze
```

2. Write most basic `bb_setup.py`

```
from bbfreeze import Freezer

freezer = Freezer(distdir='dist')
freezer.addScript('foobar.py', gui_only=True)
freezer()
```

Nota: This will work for the most basic one file scripts. For more advanced freezing you will have to provide include and exclude paths like so:

```
freezer = Freezer(distdir='dist', includes=['my_code'], excludes=['docs'])
```

---

3. (Optionally) include icon

```
freezer.setIcon('my_awesome_icon.ico')
```

4. Provide the Microsoft Visual C++ runtime DLL for the freezer. It might be possible to append your `sys.path` with the Microsoft Visual Studio path but I find it easier to drop `msvcp90.dll` in the same folder where your script resides.

5. Freeze!

```
$ python bb_setup.py
```

py2exe

Prerequisite is to install *Python on Windows*. The last release of py2exe is from the year 2014. There is not active development.

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

5.3.4 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.

5.3.5 Linux

bbFreeze

> **Aviso:** bbFreeze will ONLY work in Python 2.x environment, since it's no longer being maintained as stated by it's former maintainer. If you're interested in it, check the repository in here.

bbFreeze can be used with all distributions that has Python installed along with pip2 and/or easy_install.

For pip2, use the following:

```
$ pip2 install bbfreeze
```

Or, for easy_install:

```
$ easy_install bbfreeze
```

With bbFreeze installed, you're ready to freeze your applications.

Let's assume you have a script, say, "hello.py" and a module called "module.py" and you have a function in it that's being used in your script. No need to worry, you can just ask to freeze the main entrypoint of your script and it should freeze entirely:

```
$ bbfreeze script.py
```

With this, it creates a folder called dist/, of which contains the executable of the script and required .so (shared objects) files linked against libraries used within the Python script.

Alternatively, you can create a script that does the freezing for you. An API for the freezer is available from the library within:

```
from bbfreeze import Freezer

freezer = Freezer(distdir='dist')
freezer.addScript('script.py', gui_only=True) # Enable gui_only kwarg for app that␣
↪uses GUI packages.
freezer()
```

PyInstaller

PyInstaller can be used in a similar fashion as in OS X. The installation goes in the same manner as shown in the OS X section.

Don't forget to have dependencies such as Python and pip installed for usage.

CAPÍTULO 6

Notas adicionais

This part of the guide, which is mostly prose, begins with some background information about Python, and then focuses on next steps.

6.1 Introdução

Baseado no ‘site oficial do Python <http://python.org/about/>’_:

Python é uma linguagem de propósito geral e de alto nível, similar ao Perl, Tcl, Scheme ou Java. Algumas de umas funções principais incluem:

- **sintaxe muito clara e legível**

  A filosofia do Python foca na facilidade de leitura, desde blocos de códigos delineados com significativos espaços em branco até palavras-chave intuitivas no lugar de pontuação ilegível.

- **Extensas bibliotecas padrão e módulos de terceiros para virtualizar qualquer tarefa**

  Python é, às vezes, descrito com as palavras ‘’baterias inclusas” devido suas extensas bibliotecas padrão, as quais incluem módulos para expressões regulares, I/O de arquivos, manipulação de frações, serialização de objetos, e muito mais.

  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.

- **Integração com outros sistemas**

  Python pode ser integrado com bibliotecas Java, ativando-o para ser utilizado com os ricos ambientes em Java que programadores corporativos estão acostumados. Pode também ser estendido por módulos C ou C++ <http://docs.python.org/extending/>‘_ quando a velocidade é essencial.

- **Ubiquidade em computadores**

Python está disponível no Windows, *nix e Mac. Ele roda em qualquer lugar que uma máquina virtual do Java roda, e a implementação referencial CPython ajuda a trazer o Python a qualquer lugar em que haja um compilador C.

- Comunidade amigável

  Python possui uma vibrante e grande *comunidade* que mantém wikis, conferências, incontáveis repositórios, mailing lists, canais IRC, e muito mais. Caramba, a comunidade Python está ajudando até mesmo na escrita desse guia!

6.1.1 Sobre este guia

Propósito

O guia dos Mochileiros de Python existe para disponibilizar, para desenvolvedores novos e experientes em Python, um manual de boas práticas para a instalação, configuração e uso de Python em uma base diária.

Pela comunidade

Esse guia é arquitetado e mantido por Kenneth Reitz de uma forma aberta. Esse é um esforço dirigido à comunidade que serve a um propósito: servir a comunidade.

Para a comunidade

Todas as contribuições para o Guia são bem-vindas, de Pythonistas de todos os níveis. Se você acha que existe uma lacuna no que é coberto pelo Guia, faça um fork do Guia no GitHub e submeta um pull request.

Contribuições são bem-vindas por todos, mesmo Pythonistas de longa data ou de primeira viagem, os autores do Guia irão ajudar, com prazer, caso você tenha qualquer questão sobre adequação, integridade, ou acurácia de uma contribuição.

Para começar a trabalhar com o Guia dos Mochileiros de Python, veja a página *Contribua*.

6.2 A comunidade

6.2.1 BDFL

Guido van Rossum, o criador do Python, é apelidado de "BDFL" - "the Benevolent Dictator for Life"*

6.2.2 Fundação de Software de Python

A missão da Fundação de Software de Python é promover, proteger e avançar a linguagem de programação Python, e ajudar e facilitar o crescimento de uma comunidade diversificada e internacional de programadores Python.

Aprenda mais sobre a PSF <http://www.python.org/psf/>

6.2.3 PEPs

PEPs são "Propostas de Enriquecimento do Python". Elas descrevem mudanças ao próprio Python, ou às normas do mesmo.

Existem três diferentes tipos de PEPs (como definido pelo **PEP 1**):

"Normas" Descreve um novo recurso ou implementação.

"informacional" Descreve um problema de design, diretrizes gerais ou informação à comunidade.

"Processo" Descreve um processo relacionado ao Python.

PEPs notáveis

Existem algumas PEPs que podem ser consideradas como leitura obrigatória:

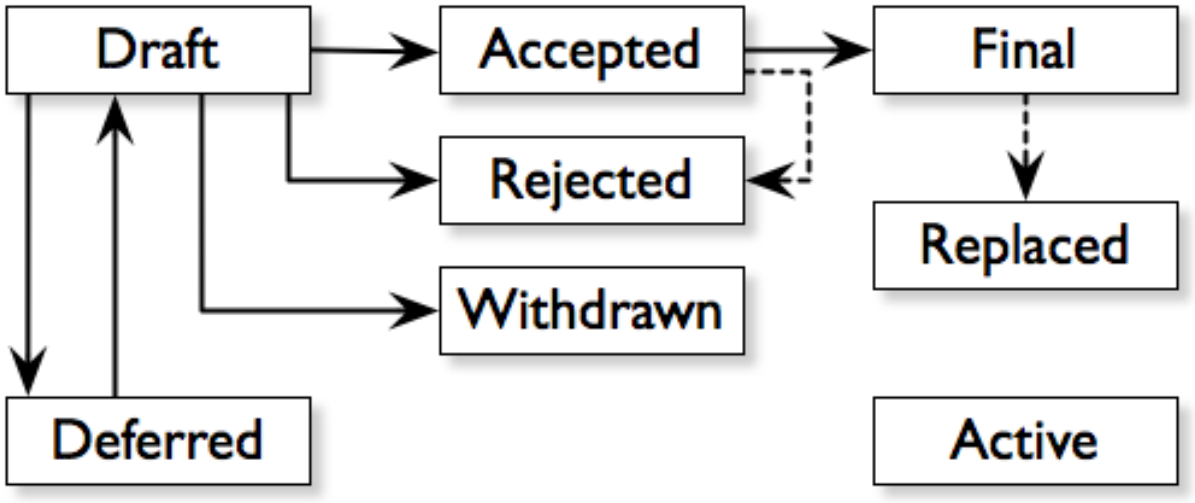
- **PEP 8: O Guia de Estilo do Python** Leia o guia. Todo ele. Siga ele.
- **PEP 20: O Zen do Python** Uma lista de 19 enunciados que explicam brevemente a filosofia por trás do Python.
- **PEP 257: Convenções de Docstring** Dá diretrizes para semântica e convenções associadas com **docstrings** do Python.

Você pode ler mais no The PEP Index*. *NT: "O índice de PEPs"

Enviando uma PEP

PEPs são avaliadas pelo público e aceitas/rejeitas depois de muita discussão. Qualquer um pode escrever e enviar uma PEP para avaliação.

Aqui está uma visão geral do fluxo de aceitação de uma PEP:



6.2.4 Conferências de Python

Os principais eventos para a comunidade Python são as conferências de desenvolvedores. As duas conferências mais notáveis são a PyCon, que acontece nos Estados Unidos, e sua parente européia, a EuroPython.

Uma lista completa de conferências é mantida em pycon.org.

6.2.5 Grupos de Usuários de Python

Grupos de usuários são onde um monte de desenvolvedores Python se encontram para apresentar ou conversar sobre tópicos de interesses relacionados ao Python. Uma lista de grupos de usuários locais é mantida na **'Python Software Foundation Wiki*<http://wiki.python.org/moin/LocalUserGroups>'_**. *NT: "Wikipédia da Fundação de Software de Python"

6.2.6 Online Communities

PythonistaCafe is an invite-only, online community of Python and software development enthusiasts helping each other succeed and grow. Think of it as a club of mutual improvement for Pythonistas where a broad range of programming questions, career advice, and other topics are discussed every day.

6.2.7 Python Job Boards

Python Jobs HQ is a Python job board, by Python Developers for Python Developers. The site aggregates Python job postings from across the web and also allows employers to post Python job openings directly on the site.

6.3 Aprendendo Python

6.3.1 Iniciante

O Tutorial Python

Este é o tutorial oficial. Ele cobre toda a parte básica e oferece um tour da linguagem e da biblioteca principal. Recomendado para aqueles que necessitam de um guia de iniciação rápida para a linguagem.

The Python Tutorial

Real Python

Real Python is a repository of free and in-depth Python tutorials created by a diverse team of professional Python developers. At Real Python you can learn all things Python from the ground up. Everything from the absolute basics of Python, to web development and web scraping, to data visualization, and beyond.

Real Python

Python Basics

pythonbasics.org is an introductory tutorial for beginners. The tutorial includes exercises. It covers the basics and there are also in-depth lessons like object oriented programming and regular expressions.

Python basics

Python para iniciantes

thepythonguru.com is a tutorial focused on beginner programmers. It covers many Python concepts in depth. It also teaches you some advanced constructs of Python like lambda expressions and regular expressions. And last it finishes off with the tutorial "How to access MySQL db using Python"

Python for Beginners

Tutorial Interativo Learn Python

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

Python for You and Me

Se você quer um livro mais tradicional, *Python For You and Me* é um excelente recurso para aprender todos os aspectos da linguagem.

Python for You and Me

Learn Python Step by Step

Techbeamers.com provides step-by-step tutorials to teach Python. Each tutorial is supplemented with logically added coding snippets and equips with a follow-up quiz on the subject learned. There is a section for Python interview questions to help job seekers. You can also read essential Python tips and learn best coding practices for writing quality code. Here, you'll get the right platform to learn Python quickly.

Learn Python Basic to Advanced

Python Tutor Online

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.

Python Tutor Online

Invente Seus Próprios Jogos de Computador com Python

Esse livro de iniciantes é para aqueles sem nenhuma experiência com programação. Cada capítulo possui o código fonte de um pequeno jogo, utilizando esses programas de exemplo para demonstrar conceitos de programação, dando ao leitor uma ideia do que um programa "parece".

invente seus próprios jogos de computador com Python

Hackeando Cifras Secretas com Python

Este livro ensina programação em Python e criptografia básica para iniciantes completos. Os capítulos disponibilizam o código fonte de várias cifras, assim como os programas que podem quebrá-las.

Hackeando Cifras Secretas com Python

Aprenda Python da Maneira Mais Difícil

Esse é um excelente guia de programação para iniciantes em Python. Ele cobre o "hello world" do console até a web.

Learn Python the Hard Way

Mergulhando em Python

Também conhecido como *Python para Programadores com 3 horas*, esse guia dá a desenvolvedores experientes com outras linguagens um curso intensivo de Python.

Crash into Python

Mergulhando em Python 3

Mergulhando em Python 3 é um bom livro para aqueles prontos para pular pra dentro do Python 3. É uma boa leitura se você está mudando de Python 2 para 3 ou se você já tem alguma experiência em programação com outra linguagem.

Dive Into Python 3

Pense Python: Como Pensar como um Cientista da Computação

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.

Enquanto explora as várias características disponíveis na linguagem Python, o autor entrelaça vários padrões de projeto e boas práticas.

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 programming and Markov Analysis.

Pense Python

Python Koans

Python Koans is a port of Edgecase's Ruby Koans. It uses a test-driven approach 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.

Para aqueles acostumados a linguagens e a descobrir puzzles por conta própria, pode ser uma opção atrativa e divertida. Para aqueles novos em Python e em programação, ter um recurso ou referência adicional será útil.

> Python Koans

Mais informações sobre desenvolvimento orientado a testes podem ser encontrados nesses recursos:

> Test Driven Development

Um Byte de Python

Um livro grátis introdutório que ensina Python em um nível iniciante, sem que nenhuma experiência anterior em programação seja necessária.

> A Byte of Python for Python 2.x A Byte of Python for Python 3.x

Computer Science Path on Codecademy

A Codecademy course for the absolute Python beginner. This free and interactive course provides and teaches the basics (and beyond) of Python programming while testing the user's knowledge in between progress. This course also features a built-in interpreter for receiving instant feedback on your learning.

> Computer Science Path on Codecademy

Code the blocks

Code the blocks provides free and interactive Python tutorials for beginners. It combines Python programming with a 3D environment where you "place blocks" and construct structures. The tutorials teach you how to use Python to create progressively more elaborate 3D structures, making the process of learning Python fun and engaging.

> Code the blocks

6.3.2 Intermediário

Python Tricks: The Book

Discover Python's best practices with simple examples and start writing even more beautiful + Pythonic code. *Python Tricks: The Book* shows you exactly how.

You'll master intermediate and advanced-level features in Python with practical examples and a clear narrative.

> Python Tricks: The Book

Python Efetivo

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 common adaptations programmers need to make to become efficient intermediate level Python programmers.

Effective Python

6.3.3 Avançado

Pro Python

Esse livro é indicado para programadores em Python intermediários ou avançados, que estão procurando entender como e porque Python funciona da maneira que funciona, e como levar seus códigos para um proximo nível.

Pro Python

Programação em Python Expert

Programação em Python Expert lida com melhores práticas de programação em Python e tem seu foco no público mais avançado.

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

Um Guia para os Métodos Mágicos de Python

Essa é uma coleção de posts de blog escrita por Rafe Kettler que explicam os "métodos mágicos" em Python. Métodos mágicos são acompanhados de dois underlines (ex: __init__) e podem fazer classes e objetos se comportarem de maneira mágica.

Um Guia para os Métodos Mágicos do Python

Nota: 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)

6.3.4 Para engenheiros e cientistas

A Primer on Scientific Programming with Python

A Primer on Scientific Programming with Python*, escrita por Hans Petter Langtangen, cobre majoritariamente a utilização de Python no meio científico. No livro, exemplos são escolhidos das ciências naturais e exatas.

A Primer on Scientific Programming with Python

Numerical Methods in Engineering with Python

Numerical Methods in Engineering with Python*, escrito por Jaan Klusalaas, enfatiza nos métodos numéricos e em como implementa-los no Python.

Numerical Methods in Engineering with Python

6.3.5 Miscellaneous Topics

Problem Solving with Algorithms and Data Structures

Problem Solving with Algorithms and Data Structures* cobre uma gama de estruturas de dados e algorítimos. Todos os conceitos são ilustrados com código Python, utilizando exemplos interativos que podem ser executados diretamente do navegador.

Problem Solving with Algorithms and Data Structures

Programming Collective Intelligence

Programming Collective Intelligence introduz a uma ampla lista de métodos de aprendizado de máquina e mineração de dados. A exposição não é muito matemáticamente formal, mas no entanto foca em explicar a intuição subjacente e mostra como implementar os algoritmos em python.

Programming Collective Intelligence

Transformando código em Python belo e idiomático

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 web server, 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

PythonistaCafe

PythonistaCafe is an invite-only, online community of Python and software development enthusiasts helping each other succeed and grow. Think of it as a club of mutual improvement for Pythonistas where a broad range of programming questions, career advice, and other topics are discussed every day.

PythonistaCafe

6.3.6 Referências

Python in a Nutshell

Python in a Nutshell, written by Alex Martelli, covers most cross-platform Python usage, from its syntax to built-in libraries to advanced topics such as writing C extensions.

Python in a Nutshell

A Referência da Linguagem Python

This is Python's reference manual. It covers the syntax and the core semantics of the language.

A Referência da Linguagem Python

Python Essential Reference

Python Essential Reference, escrito por David Beazley, é a referência definitiva para Python. Ele explica de forma concisa tanto o básico da linguagem quanto partes essenciais da biblioteca padrão. Ele cobre Python nas versões 3 e 2.6.

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+

6.4 Documentação

6.4.1 Documentação oficial

A documentação oficial da linguagem Python e das sua bibliotecas podem ser encontradas aqui:

- Python 2.x
- Python 3.x

6.4.2 *Read the Docs*

Read the Docs - Leia os Documentos - é um projeto popular da comunidade que arquiva a documentação para software de código aberto. Ele possui documentação para muitos módulos do Python, tanto populares quanto exóticos.

Read the Docs

6.4.3 pydoc

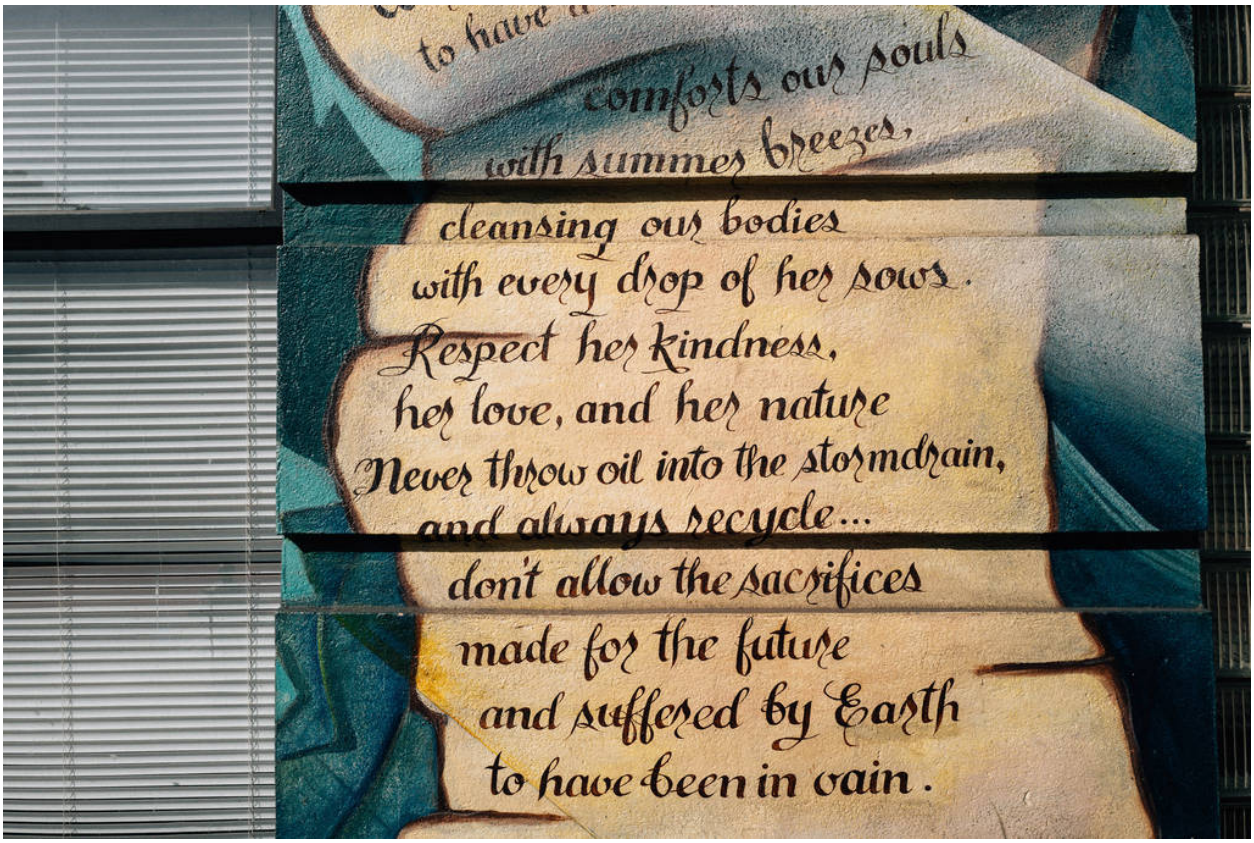
pydoc is a utility that is installed when you install Python. It allows you to quickly retrieve and search for documentation from your shell. For example, if you needed a quick refresher on the `time` module, pulling up documentation would be as simple as:

```
$ pydoc time
```

The above command is essentially equivalent to opening the Python REPL and running:

```
>>> help(time)
```

6.5 Notícias



6.5.1 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

6.5.2 Real Python

At Real Python you can learn all things Python from the ground up, with weekly free and in-depth tutorials. Everything from the absolute basics of Python, to web development and web scraping, to data visualization, and beyond.

Real Python

6.5.3 Planeta Python

Este é um compilado de novidades sobre Python vindo de um crescente número de desenvolvedores.

Planet Python

6.5.4 /r/python

/r/python é a comunidade Python no Reddit, onde usuários contribuem e votam sobre as novidades acerca de Python.

/r/python

6.5.5 Talk Python Podcast

The #1 Python-focused podcast covering the people and ideas in Python.

Talk Python To Me

6.5.6 Python Bytes Podcast

A short-form Python podcast covering recent developer headlines.

Python Bytes

6.5.7 Python Weekly

Python Weekly é um folhetim semanal gratuito e curado mostrando novidades, artigos, novos lançamentos, ofertas de emprego, etc, relacionados a Pythonis.

Python Weekly

6.5.8 *Python News*

Python News é a seção de notócias to site oficial de Python (www.python.org). Ela brevemenente destaca as novidades da comunidade Python.

Python News

6.5.9 Importe *Python Weekly*

Weekly Python Newsletter containing Python Articles, Projects, Videos, and Tweets delivered in your inbox. Keep Your Python Programming Skills Updated.

Importe Python Weekly Newsletter

6.5.10 *Awesome Python Newsletter*

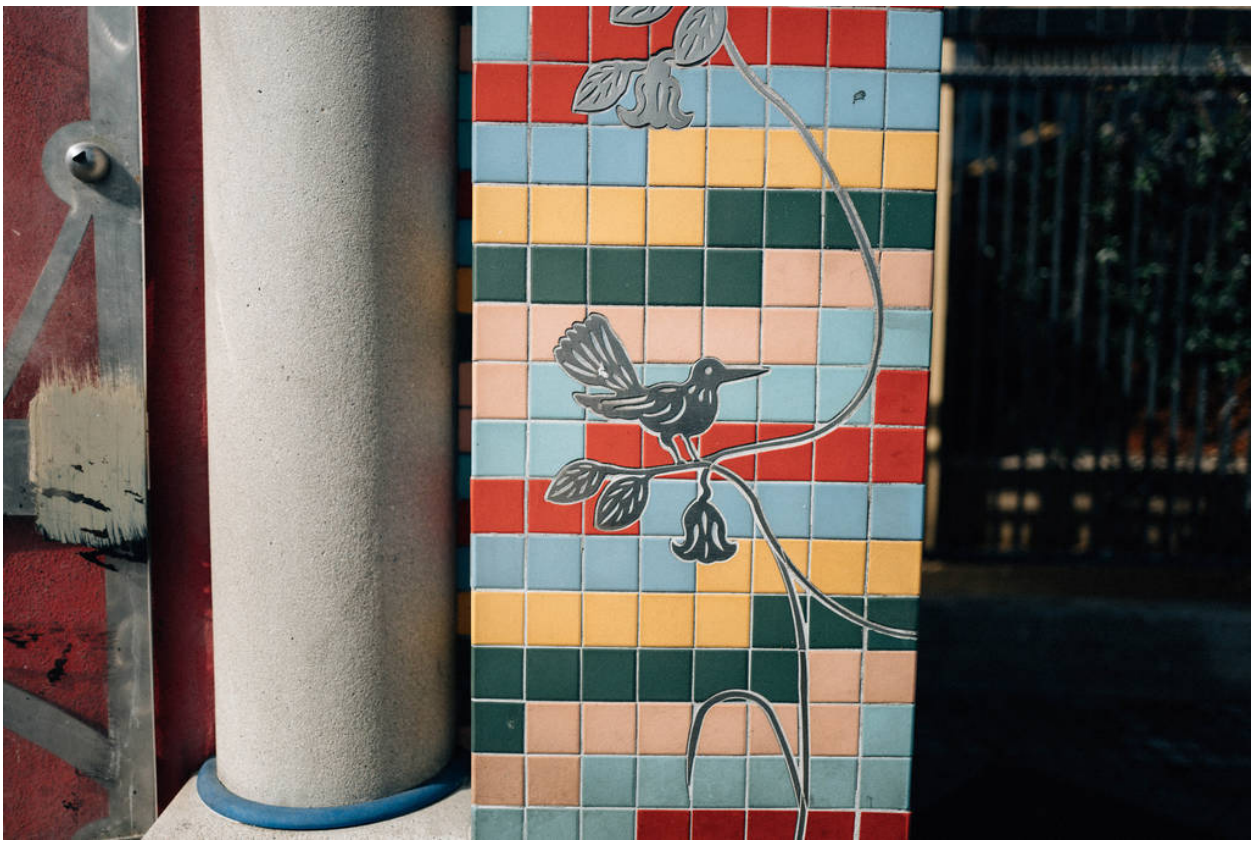
A weekly overview of the most popular Python news, articles, and packages.

Awesome Python Newsletter

Nota: Todas as notas definidas nas escalas musicais diatônicas e cromáticas foram intencionalmente excluidas dessa lista de notas adicionais.

Notas de contribuição e informações legais (aos interessados)

6.6 Contribua



O Guia de Python está em ativo desenvolvimento, e contribuições são bem-vindas.

Se você tem alguma solicitação, sugestão, ou deseja reportar erros, por favor abra um novo 'issue' em nossa página no Github. Para enviar correções, por favor, nos envie um 'pull request' no Github. Você também pode nos contactar diretamente via Github. Uma vez que suas modificações forem incorporadas, você automaticamente será adicionado a Lista de Contribuidores.

6.6.1 Guia de Estilo

Para todas as contribuições, por favor siga o ref:*guide-style-guide*.

6.6.2 Lista de tarefas

Se você deseja contribuir, tem muito o que ser feito. Aqui vai uma pequena todo list.

- Estabelecer os casos para recomendar "use isso" vs "alternativas são. . . "

Por fazer: Write about Blueprint

(A entrada original está localizada na /home/docs/checkouts/readthedocs.org/user_builds/python-guide-pt-br/checkouts/latest/docs/scenarios/admin.rst, linha 386.)

Por fazer: Fill in "Freezing Your Code" stub

(A entrada original está localizada na /home/docs/checkouts/readthedocs.org/user_builds/python-guide-pt-br/checkouts/latest/docs/shipping/freezing.rst, linha 42.)

Por fazer: Replace this kitten with the photo we want.

(A entrada original está localizada na /home/docs/checkouts/readthedocs.org/user_builds/python-guide-pt-br/checkouts/latest/docs/shipping/publishing.rst, linha 8.)

Por fazer: Incluir exemplos de código demonstrativos de cada um dos projetos listados. Explicar por que o mesmo é um código excelente. Use exemplos complexos.

(A entrada original está localizada na /home/docs/checkouts/readthedocs.org/user_builds/python-guide-pt-br/checkouts/latest/docs/writing/reading.rst, linha 50.)

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

(A entrada original está localizada na /home/docs/checkouts/readthedocs.org/user_builds/python-guide-pt-br/checkouts/latest/docs/writing/reading.rst, linha 52.)

6.7 Licença

Este guia é licenciado pelas normas Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license.

6.8 The Guide Style Guide

Como toda documentação, ter um formato consistente ajuda a fazer o documento mais claro. Com o objetivo de fazer O Guia ser de mais fácil digestão, todas as contribuições devem seguir as regras de estilo descritas aqui onde necessário.

The Guide is written as *reStructuredText*.

Nota: 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

---

Nota: On any page of the rendered HTML you can click "Show Source" to see how authors have styled the page.

6.8.1 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.

6.8.2 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
---------
```

6.8.3 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 Standard American English, not British English.

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.

6.8.4 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 for Unix console examples.

For Windows console examples, use `doscon` or `powershell` instead of `console`, and omit the `$` prefix.

Python interpreter examples:

```
Label the example::

.. code-block:: python

    >>> import this
```

Python examples:

```
Descriptive title::

.. code-block:: python

    def get_answer():
        return 42
```

6.8.5 Externally Linking

- Prefer labels for well known subjects (e.g. proper nouns) when linking:

```
Sphinx_ is used to document Python.

.. _Sphinx: https://www.sphinx-doc.org
```

- Prefer to use descriptive labels with inline links instead of leaving bare links:

```
Read the `Sphinx Tutorial <https://www.sphinx-doc.org/en/master/usage/quickstart.
↪html>`_
```

- Avoid using labels such as "click here", "this", etc., preferring descriptive labels (SEO worthy) instead.

6.8.6 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
------------
```

6.8.7 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
```

6.8.8 TODOs

Please mark any incomplete areas of The Guide with a todo directive. To avoid cluttering the *Lista de tarefas*, 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
```

Índice