> 译者按：原文写于2011年末，虽然文中关于Python 3的一些说法可以说已经不成立了，但是作为一篇面向从其他语言转型到Python的程序员来说，本文对Python的生态系统还是做了较为全面的介绍。文中提到了一些第三方库，但是Python社区中强大的第三方库并不止这些，欢迎各位Pytonistas补充。
>
> - 原文链接：http://mirnazim.org/writings/python-ecosystem-introduction/
> - 译文链接：http://codingpy.com/article/python-ecosystem-introduction/

开发者从PHP、Ruby或其他语言转到Python时，最常碰到的第一个障碍，就是缺乏对Python生态系统的全面了解。开发者经常希望能有一个教程或是资源，向他们介绍如何以大致标准的方式完成大部分的任务。

本文中的内容，是对我所在公司内部维基百科的摘录，记录了面向网络应用开发的Python生态系统基础知识，目标受众则是公司实习生、培训人员以及从其他语言转型到Python的资深开发者。

文中所列的内容或资源是不完整的。我也打算把它当成一项一直在进行中的工作（work in perpetual progress）。希望经过不断的增补修订，本文会成为Python生态系统方面的一篇详尽教程。

目标受众

本文的目的，不是教大家Python编程语言。读完这篇教程，你也不会瞬间变成一名Python高手。我假设大家已经有一定的Python基础。如果你是初学者，那么别再继续读下去了。先去看看Zed Shaw所写的《笨办法学Python》，这是本质量很高的免费电子书，看完之后再回头阅读这篇教程吧。

我假设你们使用的是Linux（最好是Ubuntu/Debian）或是类Linux操作系统。为什么？因为这是我最熟悉的系统。我在Windows平台或Mac OS X平台上，没有专业的编程经验，只是测试过不同浏览器的兼容性。如果你用的是这两个平台，那么请参考下面的资料安装Python。

- Python 101: Setting up Python on Windows
- Official documentation for Python on Windows
- Official documentation for Python on Mac OS X

你还可使用搜索引擎，查找你使用的操作系统中安装Python的最好方法。如果你有什么疑问，我强烈建议你去Stack Overflow平台上提问。

该选择哪个版本？

Python 2.x是目前的主流；Python 3是崭新的未来。如果你不关心这个问题，可以直接跳到下面的Python安装部分。（译者注：原文作者写这篇文章时是2011年，当时Python 3才发展没几年。）

刚接触Python的话，安装3.x版本看上去是很自然的第一步，但是这可能并不是你想要的。

目前有两个积极开发中的Python版本——2.7.x与3.x（也被称为Python 3，Py3K和Python 3000）。Python 3是一个不同于Python 2的语言。二者在语义、语法上，既存在细微的区别，又有着截然不同的一面。截至今天，Python2.6/2.7是安装数量和使用度最高的版本。许多主流的Python库、框架、工具都没有做到100%兼容Python 3。

因此，最稳妥的选择就是使用2.x版（更准确的说，即2.7.x）。务必只在你需要或者完全了解情况的前提下，才选择Python 3。

Python 3 Wall of Shame网站记录了Python 3对各种库的兼容情况。在使用Python 3之前，仔细查阅下这个网站的内容。

> 译者注：现在，主流第三方库和框架对Python 3的支持度已经很高。根据py3readiness网站的统计，360个最受欢迎的Python包中已经有315个支持Python 3。具体的支持情况，可以查看这个网站。一定程度上说，Python 3已经成为新的主流。

使用哪种虚拟机

Python的解释器，又叫做Python虚拟机，它有多种不同的实现。其中，主流实现方式是CPython，装机量也最高，同时也是其他虚拟机的参考实现。

PyPy是利用Python语言实现的Python；Jython则使用Java实现，并运行在Java虚拟机之上；IronPython是用.NET CLR实现的Python。

除非真的有重大理由，否则应该选择CPython版本的实现，避免出现意外情况。

如果这些有关版本和虚拟机的唠叨话让你读了头疼，那你只需要使用CPython 2.7.x即可。

Python安装

大部分Linux/Unix发行版和Mac OS X都预装了Python。如果你没有安装或者已有的版本比较旧，那么你可以通过下面的命令安装2.7.x版：

Ubuntu/Debian及其衍生系统

```
$ sudo apt-get install python2.7
```

`sudo`是类Unix系统中的一个程序，可以让用户以其他用户的权限（通常是超级用户或root用户）运行程序。

Fedora/Red Hat及类似系统

```
sudo yum install python2.7
```

在RHEL（Red Hat Enterprise Linux的缩写）平台上，你可能需要启用EPEL

软件源（repositories），才能正常安装。

在本文后面的示例中，我会使用`sudo`程序；你应将其替换为自己版本中的相应命令或程序。

理解Python的包（package）

首先你需要了解的是，Python没有默认的包管理工具。事实上，Python语言中包的概念，也是十分松散的。

你可能也知道，Python代码按照模块（module）划分。一个模块，可以是只有一个函数的单个文件，也可以是包含一个或多个子模块的文件夹。包与模块之间的区别非常小，每个模块同时也可以视作一个包。

那么模块与包之间，到底有什么区别？要想解答这个问题，你首先要了解Python是如何查找模块的。

与其他编程环境类似，Python中也有一些函数和类（比如`str`，`len`库`Exception`）是存在于全局作用域（global scope，在Python中被称为builtin scope）的，其他的函数和类则需要通过`import`语句进行引用。例如：

```
>>> import os
>>> from os.path import basename, dirname
```

这些包就在你的文件系统中的某处，所以能被`import`语句发现。那么Python是怎么知道这些模块的地址？原来，在你安装Python虚拟机的时候，就自动设置了这些地址。当然平台不同，这些地址也就不一样。

你可以通过`sys.path`查看系统中的包路径。这是我的笔记本运行该命令之后的输出结果，系统是Ubuntu 11.10 Oneric Ocelot。

```
>>> import sys
>>> print sys.path
['',
 '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-linux2',
 '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old',
 '/usr/lib/python2.7/lib-dynload',
 '/usr/local/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages/PIL',
 '/usr/lib/python2.7/dist-packages/gst-0.10',
 '/usr/lib/python2.7/dist-packages/gtk-2.0',
 '/usr/lib/pymodules/python2.7',
 '/usr/lib/python2.7/dist-packages/ubuntu-sso-client',
 '/usr/lib/python2.7/dist-packages/ubuntuone-client',
 '/usr/lib/python2.7/dist-packages/ubuntuone-control-panel',
 '/usr/lib/python2.7/dist-packages/ubuntuone-couch',
 '/usr/lib/python2.7/dist-packages/ubuntuone-installer',
 '/usr/lib/python2.7/dist-packages/ubuntuone-storage-protocol']
```

这行代码会告诉你Python搜索指定包的所有路径，这个路径就存储在一个Python列表数据类型中。它会先从第一个路径开始，一直往下检索，直到找到匹配的路径名。这意味着，如果两个不同的文件夹中包含了两个同名的

包，那么包检索将会返回其遇到的第一个绝对匹配地址，不会再继续检索下去。

你现在可能也猜到了，我们可以轻松地修改（hack）包检索路径，做到你指定的包第一个被发现。你只需要运行下面的代码：

```
>>> sys.path.insert(0, '/path/to/my/packages')
```

尽管这种做法在很多情况下十分有用，但是你必须牢记`sys.path`很容易被滥用。务必在必要时才使用这种方法，并且不要滥用。

`site`模块控制着包检索路径设置的方法。每次Python虚拟机初始化时，就会被自动引用。如果你想更详细地了解整个过程，可以查阅官方文档。

PYTHONPATH环境变量

`PYTHONPATH`是一个可以用来增强默认包检索路径的环境变量。可以把它看作是一个`PATH`变量，但是一个只针对Python的变量。它只是一些包含有Python模块的文件路径列表（不是`sys.path`所返回的Python列表），每个路径之间以：分隔。设置方法很简单，如下：

```
export PYTHONPATH=/path/to/some/directory:/path/to/another/directory:/path/to/y
```

在某些情况下，你不用覆盖已有的`PYTHONPATH`，只需要在开头或结尾加上新的路径即可。

```
export PYTHONPATH=$PYTHONPATH:/path/to/some/directory    # Append
export PYTHONPATH=/path/to/some/directory:$PYTHONPATH    # Prepend
```

`PYTHONPATH`、`sys.path.insert`和其他类似的方法，都是hack小技巧，一般情况下最好不要使用。如果它们能够解决本地开发环境出现的问题，可以使用，但是你的生产环境中不应该依赖这些技巧。要取得同样的效果，我们还可以找到更加优雅的方法，稍后我会详细介绍。

现在你明白了Python如何查找已安装的包，我们就可以回到一开始的那个问题了。Python中，模块和包的区别到底是什么？包就是一个或多个模块/子模块的集合，一般都是以经过压缩的tarball文件形式传输，这个文件中包含了：1. 依赖情况（如果有的话）；2.将文件复制到标准的包检索路径的说明；3. 编译说明——如果文件中包含了必须要经过编译才能安装的代码。就是这点区别。

第三方包（Third Party packages）

如果想利用Python进行真正的编程工作，你从一开始就需要根据不同的任务安装第三方包。

在Linux系统上，至少有3种安装第三方包的方法。

1. 使用系统本身自带的包管理器（deb, rpm等）
2. 通过社区开发的类似`pip`，`easy_install`等多种工具

3. 从源文件安装

这三种方法做的几乎是同一件事情，即安装依赖包，视情况编译代码，然后把包中模块复制到标准包检索路径。

尽管第二种和第三种方法在所有操作系统中的实现都一致，我还是要再次建议你查阅Stack Overflow网站的问答，找到你所使用系统中其他安装第三方包的方法。

去哪找第三方包？

在安装第三方包之前，你得先找到它们。查找包的方法有很多。

1. 你使用的系统自带的包管理器
2. Python包索引（也被称为PyPI）
3. 各种源码托管服务，如Launchpad， Github， Bitbucket等。

通过系统自带的包管理器安装

使用系统自带的包管理器安装，只需要在命令行输入相应命令，或是使用你用来安装其他应用的GUI应用即可。举个例子，要在Ubuntu系统上安装 `simplejson`（一个JSON解析工具），你可以输入下面的命令：

```
$ sudo apt-get install python-simplejson
```

通过pip安装

`easy_install` 已经不太受开发者欢迎。本文将重点介绍 *`easy_install`* 的替代者——*`pip`*。

`pip` 是一个用来安装和管理Python包的工具。它并不是一个Python虚拟机自带的模块，所以我们需要先安装。在Linux系统中，我一般会这样操作：

```
$ sudo apt-get install python-pip
```

在安装其他包之前，我总是会把 `pip` 升级到PyPI中的最新版本，因为Ubuntu默认源中的版本比PyPI的低。我这样升级 `pip`。

```
$ sudo pip install pip --upgrade
```

现在，你可以通过运行 `run pip install package-name`，安装任何Python包。所以，要安装 `simplejson` 的话，你可以运行以下命令：

```
$ sudo pip install simplejson
```

移除包也一样简单。

```
$ sudo pip uninstall simplejson
```

`pip` 默认会安装PyPI上最新的稳定版，但是很多时候，你会希望安装指定版本的包，因为你的项目依赖那个特定的版本。要想指定包的版本，你可以这样做：

```
$ sudo pip install simplejson==2.2.1
```

你还会经常需要升级、降级或者重装一些包。你可以通过下面的命令实现：

```
$ sudo pip install simplejson --upgrade         # Upgrade a package to the late
$ sudo pip install simplejson==2.2.1 --upgrade  # Upgrade/downgrade a package t
```

接下来，假设你想安装某个包的开发版本，但是代码没有放在PyPI上，而是在版本控制仓库中，你该怎么办？`pip`也可以满足这个需求，但是在此之前，你需要在系统上安装相应的版本控制系统（VCS）。在Ubuntu平台，你可以输入下面的命令：

```
$ sudo apt-get install git-core mercurial subversion
```

安装好VCS之后，你可以通过下面的方式从远程仓库中安装一个包：

```
$ sudo pip install git+http://hostname_or_ip/path/to/git-repo#egg=packagename
$ sudo pip install hg+http://hostname_or_ip/path/to/hg-repo#egg=packagename
$ sudo pip install svn+http://hostname_or_ip/path/to/svn-repo#egg=packagename
```

从本地仓库中安装也同样简单。注意下面文件系统路径部分的三个斜杠（///）。

```
$ sudo pip install git+file:///path/to/local/repository
```

通过`git`协议安装时，请注意，你要像下面这样使用`git+git`前缀：

```
$ sudo pip install git+git://hostname_or_ip/path/to/git-repo#egg=packagename
```

现在，你可能在纳闷这些命令中的`eggs`是什么东西？目前你只需要知道，一个`egg`就是经zip压缩之后的Python包，其中包含了包的源代码和一些元数据。`pip`在安装某个包之前，会构建相关的egg信息。你可以打开代码仓库中的`setup.py`文件，查看egg的名字（几乎都会注明）。找到`setup`部分，然后看看有没有一行类似`name="something"`的代码。你找到的代码可能会和下面这段代码类似（来自simplejson包中的`setup.py`文件）。

```
setup(
    name="simplejson", # <--- This is your egg name
    version=VERSION,
    description=DESCRIPTION,
    long_description=LONG_DESCRIPTION,
    classifiers=CLASSIFIERS,
    author="Bob Ippolito",
    author_email="bob@redivi.com",
    url="http://github.com/simplejson/simplejson",
    license="MIT License",
    packages=['simplejson', 'simplejson.tests'],
    platforms=['any'],
    **kw)
```

假如没有`setup.py`文件呢？你该怎么查找egg的名字？答案是，你根本不用去找。只要把包的源代码拷贝到你的项目文件夹，之后就可以和你自己写的代码一样引用和使用啦。

--user参数

以上所有的例子，都是在系统层面安装指定的包。如果你使用`pip install`时，加上`--user`这个参数，这些包将会安装在该用户的'~/.local`文件夹之下。例如，在我的机器上，运行效果是这样的：

```
$ pip install --user markdown2
Downloading/unpacking markdown2
  Downloading markdown2-1.0.1.19.zip (130Kb): 130Kb downloaded
  Running setup.py egg_info for package markdown2

Installing collected packages: markdown2
  Running setup.py install for markdown2
    warning: build_py: byte-compiling is disabled, skipping.

    changing mode of build/scripts-2.7/markdown2 from 664 to 775
    warning: install_lib: byte-compiling is disabled, skipping.


    changing mode of /home/mir/.local/bin/markdown2 to 775
Successfully installed markdown2
Cleaning up...
```

注意markdown2这个Python包的安装路径（`/home/mir/.local/bin/markdown2`）

不在系统层面安装所有的Python包有很多理由。稍后在介绍如何为每个项目设置单独、孤立的Python环境时，我会具体说明。

从源文件安装

从源文件安装Python包，只需要一行命令。将包文件解压，然后运行下面的命令：

```
cd /path/to/package/directory
python setup.py install
```

尽管这种安装方法与其他的方法没什么区别，但是要记住：`pip`永远是安装Python包的推荐方法，因为`pip`可以让你轻松升级/降级，不需要额外手动下载、解压和安装。从源文件安装时如果其他方法都行不通时，你的最后选择（一般不会存在这种情况）。

安装需要编译的包

虽然我们已经介绍了大部分与包安装相关的内容，仍有一点我们没有涉及：含有C/C++代码的Python包在安装、使用之前，需要先编译。最明显的例子就是数据库适配器（database adapters）、图像处理库等。

尽管`pip`可以管理源文件的编译，我个人更喜欢通过系统自带的包管理器安装这类包。这样安装的就是预编译好的二进制文件。

如果你仍想（或需要）通过`pip`安装，在Ubuntu系统下你需要执行下面的操作。

安装编译器及相关工具：

```
$ sudo apt-get install build-essential
```

安装Python开发文件（头文件等）：

```
$ sudo aptitude install python-dev-all
```

如果你的系统发行版本中没有提供`python-dev-all`，请查找名字类似`python-dev`、`python2.X-dev`的相关包。

假设你要安装`psycopg2`（PostgreSQL数据库的Python适配器），你需要安装PostgreSQL的开发文件。

```
$ sudo aptitude install  postgresql-server-dev-all
```

满足这些依赖条件之后，你就可以通过`pip install`安装了。

```
$ sudo pip install psycopg2
```

这里应该记住一点：并非所有这类包都兼容pip安装方式。但是，如果你自信可以成功编译源文件，并且（或者）已经有目标平台上的必要经验和知识，那么你完全可以按照这种方式安装。

开发环境

不同的人设置开发环境的方法也不同，但是在几乎所有的编程社区中，肯定有一种方法（或者超过一种）比其他方法的接受度更高。尽管开发环境设置的与别人不同没有问题，一般来说接受度更高的方法经受住了高强度的测试，并被证实可以简化一些日常工作的重复性任务，并且可以提高可维护性。

virtualenv

Python社区中设置开发环境的最受欢迎的方法，是通过virtualenv。Virtualenv是一个用于创建孤立Python环境的工具。那么现在问题来了：为什么我们需要孤立的Python环境？要回答这个问题，请允许我引用virtualenv的官方文档。

> 我们要解决的问题之一，就是依赖包和版本的管理问题，以及间接地解决权限问题。假设你有一个应用需要使用LibFoo V1，但是另一个应用需要V2。那么你如何使用两个应用呢？如果你把需要的包都安装在`/usr/lib/python2.7/site-packages`（或是你的系统默认路径），很容易就出现你不小心更新了不应该更新的应用。

简单来说，你的每一个项目都可以拥有一个单独的、孤立的Python环境；你可以把所需的包安装到各自孤立的环境中。

还是通过`pip`安装virutalenv。

```
$ sudo pip install virtualenv
```

安装完之后，运行下面的命令，为你的项目创建孤立的Python环境。

```
$ mkdir my_project_venv
```

```
$ virtualenv --distribute my_project_venv
# The output will something like:
New python executable in my_project_venv/bin/python
Installing distribute.............................................done.
Installing pip.....................done.
```

那么这行代码都做了些什么呢？你创建了一个名叫`my_project_venv`的文件夹，用于存储新的Python环境。`--distribute`参数告诉virtualenv使用基于`distribute`包开发的新的、更好的打包系统，而不是基于`setuptools`的旧系统。你现在只需要知道，`--distribute`参数将会自动在虚拟环境中安装`pip`，免去了手动安装的麻烦。随着你的Python编程经验和知识增加，你会慢慢明白这个过程的具体细节。

现在查看`my_project_venv`文件夹中的内容，你会看到类似下面的文件夹结构：

```
# Showing only files/directories relevant to the discussion at hand
.
|-- bin
|   |-- activate  # <-- Activates this virtualenv
|   |-- pip       # <-- pip specific to this virtualenv
|   `-- python    # <-- A copy of python interpreter
`-- lib
    `-- python2.7 # <-- This is where all new packages will go
```

通过下面的命令，激活虚拟环境：

```
$ cd my_project_venv
$ source bin/activate
```

使用`source`命令启动`activate`脚本之后，你的命令行提示符应该会变成这样：

```
(my_project_venv)$
```

虚拟环境的名称会添加在$提示符的前面。

现在运行下面的命令，关闭虚拟环境：

```
(my_project_venv)$ deactivate
```

当你在系统层面安装virtualenv时（如果激活了虚拟环境，请先关闭），可以运行下面的命令帮助自己理解。

首先，我们来看看如果我们在终端输入`python`或者`pip`，系统会使用哪个执行文件。

```
$ which python
/usr/bin/python
$ which pip
/usr/local/bin/pip
```

现在再操作一次，但是首先要激活virtualenv，注意输出结果的变化。在我的机器上，命令的输出结果时这样的：

```
$ cd my_project_venv
```

```
$ source bin/activate
(my_project_venv)$ which python
/home/mir/my_project_venv/bin/python
(my_project_venv)$ which pip
/home/mir/my_project_venv/bin/pip
```

`virtualenv`所做的，就是拷贝了一份Python可执行文件，然后创建了一些功能脚本以及你在项目开发期间用于安装、升级、删除相关包的文件夹路径。它还施展了一些包检索路径/PYTHONPATH魔法，确保实现以下几点：1. 在你安装第三方包时，它们被安装在了当前激活的虚拟环境，而不是系统环境中；2. 当在代码中引用第三方包时，当前激活的虚拟环境中的包将优先于系统环境中的包。

这里有很重要的一点要注意：系统Python环境中安装的所有包，默认是可以在虚拟环境中调用的。这意味着，如果你在系统环境中安装了`simplejson`包，那么所有的虚拟环境将自动获得这个包的地址。你可以在创建虚拟环境时，通过添加`--no-site-packages`选项，取消这个行为，就像这样：

```
$ virtualenv my_project_venv --no-site-packages
```

virtualenvwrapper

`virtualenvwrapper`是`virtualenv`的封装器（wrapper），提供了一些非常好的功能，便捷了创建、激活、管理和销毁虚拟环境的操作，否则将会是件琐事。你可以运行如下命令安装`virtualenvwrapper`：

```
$ sudo pip install virtualenvwrapper
```

安装结束之后，你需要进行一些配置。下面是我的配置：

```
if [ `id -u` != '0' ]; then
  export VIRTUALENV_USE_DISTRIBUTE=1        # <-- Always use pip/distribute
  export WORKON_HOME=$HOME/.virtualenvs       # <-- Where all virtualenvs will
  source /usr/local/bin/virtualenvwrapper.sh
  export PIP_VIRTUALENV_BASE=$WORKON_HOME
  export PIP_RESPECT_VIRTUALENV=true
```

这些配置中，唯一必须的是`WORKON_HOME`与`source /usr/local/bin/virtualenvwrapper.sh`。其他的配置则是根据我的个人偏好进行的。

将上面的配置添加到`~/.bashrc`文件的最后，然后在你当前打开的终端界面中运行下面的命令：

```
$ source ~/.bashrc
```

关掉所有打开的终端窗口和Tab窗口，也能取得同样地效果。当你再次打开终端窗口或Tab窗口时，`~/.bashrc`将会被执行，自动设置好你的`virtualenvwrapper`。

现在如果想创建、激活、关闭或是删除虚拟环境，你可以运行下面的代码：

```
$ mkvirtualenv my_project_venv
$ workon my_project_venv
```

```
$ deactivate
$ rmvirtualenv my_project_venv
```

virtualenvwrapper还支持tab自动补全功能。

你可以前往virtualenvwrapper项目主页查看更多命令和配置选项。

通过pip和virtualenv进行基本的依赖包管理

`pip`与`virtualenv`结合使用，可以为项目提供基本的依赖包管理功能。

你可以使用`pip freeze`导出目前安装的包列表。例如，下面就是我用来开发这个博客网站所用的Python包：

```
$ pip freeze -l
Jinja2==2.6
PyYAML==3.10
Pygments==1.4
distribute==0.6.19
markdown2==1.0.1.19
```

注意，我使用了-l选项。它告诉`pip`只导出当前激活的虚拟环境中安装的包，忽略全局安装的包。

你可以将导出的列表保存至文件，并把文件添加到版本控制系统（VCS）。

```
$ pip freeze -l  > requirements.txt
```

通过`pip`，我们可以从写入了`pip freeze`命令结果的文件中，安装相应的包。

其他重要工具

前面我们介绍了有关Python版本、虚拟机和包管理的基础知识，但是日常工作中还有其他任务需要使用专门的工具来完成。虽然我无法详细介绍每一个工具，我会尽量做一个大概的介绍。

提前说声对不起，因为下面介绍的大部分工具都是与网络应用开发相关的。

编辑器

提供在Python中进行编程的优秀编辑器有很多。我个人倾向于Vim，但是我不想引发一场编辑器优劣大战。

对Python编程支持较好地编辑器和集成开发环境（IDEs），主要有Vim/GVim， Emacs, GNOME主题下的GEdit, Komodo Edit, Wing IDE, PyCharm等。还有其他编辑器，但是上面列举的这些应该是最受欢迎的。你应该选择最适合自己的工具。

Pyflakes：源码检查

Pyflakes是一个简单的程序，通过分析文件的文本内容，检查Python源文件中的错误。它可以检查语法和部分逻辑错误，识别被引用但没有使用的模块，以及只使用了一次的变量，等等。

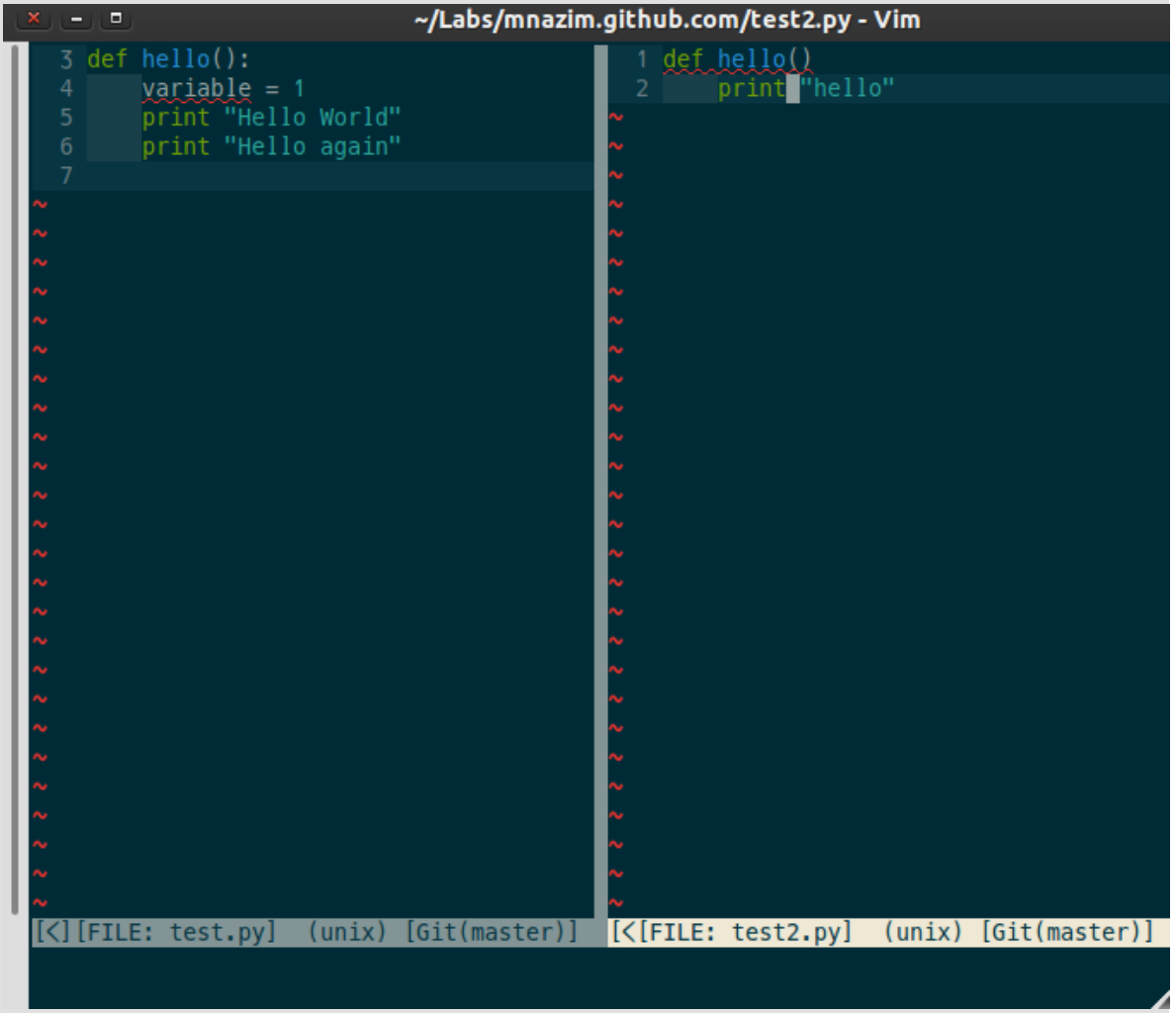
你可以通过`pip`安装：

```
$ pip install pyflakes
```

然后像下面那样，在命令行调用pyflakes，传入Python源文件作为参数：

```
$ pyflakes filename.py
```

Pyflakes还可以嵌入到你的编辑器中。下面这张图显示的是嵌入了Vim之后的情况。注意出现了红色的波浪线。



你可以在Stack Overflow上咨询如何在你使用的编辑器重添加Pyflakes支持。

Requests：为人类开发的HTTP库

Requests库让你轻轻松松使用HTTP协议。

首先通过`pip`安装：

```
$ pip install requests
```

下面是一个简单的使用示例：

```
>>> import requests
>>> r = requests.get('https://api.github.com', auth=('user', 'pass'))
```

```
>>> r.status_code
204
>>> r.headers['content-type']
'application/json'
>>> r.content
...
```

更多详情，请查看Requests的文档。

Flask：网络开发微框架

Flask是一个基于Werkzeug与Jinja2这两个库的Python微框架。

首先通过`pip`安装：

```
$ pip install Flask
```

下面是一个简单的使用示例：

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

这样运行Flask应用：

```
$ python hello.py
 * Running on http://localhost:5000/
```

Flask官网

Django：面向网络开发的全栈框架

Django是一个全栈网络框架。它提供了ORM、HTTP库、表格处理、XSS过滤、模板引擎以及其他功能。

这样通过`pip`安装：

```
$ pip install Django
```

前往Django官网，跟着教程学习即可。非常简单。

Fabric：简化使用SSH部署网站和执行系统管理任务的方式

Fabric是一个命令行工具，可以简化使用SSH进行网站部署或执行系统管理任务的过程。

它提供了一套基本的操作，可以执行本地或远程命令行命令，上传/下载文件，以及提示用户进行输入或者取消执行等辅助性功能。

你可以通过`pip`安装：

```
$ pip install fabric
```

下面是用Fabric写的一个简单任务：

```
from fabric.api import run

def host_type():
    run('uname -s
```

接下来，你可以在一个或多个服务器上执行该任务：

```
$ fab -H localhost host_type
[localhost] run: uname -s
[localhost] out: Linux

Done.
Disconnecting from localhost... done.
```

Fabric官网

SciPy：Python中的科学计算工具

如果你的工作涉及科学计算或数学计算，那么SciPy就是必不可少的工具。

SciPy (pronounced "Sigh Pie") 是一个开源的数学、科学和工程计算包。SciPy包含的模块有最优化、线性代数、积分、插值、特殊函数、快速傅里叶变换、信号处理和图像处理、常微分方程求解和其他科学与工程中常用的计算。与其功能相类似的软件还有MATLAB、GNU Octave和Scilab。SciPy目前在BSD许可证下发布。它的开发由Enthought资助

前往SciPy官网，获取详细的下载/安装说明以及文档。

PEP 8：Python风格指南

虽然PEP 8本身不是一个工具，但毋庸置疑的是，它是Python开发方面一个非常重要的文件。

PEP 8这个文件中，定义了主流Python发行版本中标准库的编码规范。文件的唯一目的，就是确保其他的Python代码都能遵守同样地代码结构以及变量、类和函数命名规律。确保你充分了解并遵循该风格指南。

PEP 8链接

强大的Python标准库

Python的标准库内容非常丰富，提供了大量的功能。标准库中包含了众多内建模块（built-in modules，用C语言编写的），可以访问类似文件读/写（I/O）这样的系统功能，还包括了用Python编写的模块，提供了日常编程中许多问题的标准解决方案。其中一些模块的设计思路很明显，就是要鼓励和增强Python程序的可移植性，因此将平台相关的细节抽象为了不依赖于平台的API接口。

查看标准库的官方文档。

推荐阅读

David Goodger的《如何像Python高手一样编程》一文，深入介绍了许多Python的惯用法和技巧，可以立刻为你增添许多有用的工具。

Doug Hellmann的系列文章Python Module of the Week。这个系列的焦点，是为Python标准库中模块编写示例代码。

练习

我在本文中所介绍的内容，触及的还只是Python生态系统的表面。Python世界中，几乎针对每一个你能想象到的任务，都存在相关的工具、库和软件。这些明显无法在一篇文章中尽述。你必须要自己慢慢探索。

Python有伟大的社区，社区中的人很聪明，也很有耐心，乐于帮助Python语言的初学者。所以，你可以选择一个最喜欢的开源项目，去它的IRC频道找人聊天；关注邮件列表，并积极提问；和有丰富Python系统实施经验的人交谈。慢慢地，随着你的经验和知识逐步积累，你也会成为他们之中的一员。

最后，我为大家推荐Python之禅。反复回味、思考这几段话，你一定会有所启发！

```
>>> 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!
```