Python Guide Documentation

发布 ***0.0.1***

Kenneth Reitz

2017 年 08 月 18 日

Contents

地球人，你们好！欢迎来到Python漫游指南。

这是一份生动的、栩栩如生的指南。 如果您有意贡献内容, 在GitHub上fork我!

这份手写指南着眼于Python的安装、配置和日常使用，旨在为新手和专业的Python开发者在提供一份最佳实践。

这是一份充满 **自我见解** 的指南，内容与Python官方文档大相径庭。这份指南不会是简单的罗列每一个Python框架，取而代之，我们会提供一系列实用而又精炼的建议。

那么开始吧！开始之前，首先要确定你明白自己的毛巾在哪里。

译者注： 《银河系漫游指南》曾经明确指出，毛巾是对一个星际漫游者来说最有用的东西， 至于为什么，我也不知道，因为我也没看过。

CHAPTER 1

开始吧

Python菜鸟？来，先教你如何正确配置Python环境。

选择一个解释器

Python的现状 (3 & 2)

当选择Python解释器的时候，总会面临一个问题：“我该使用Python 2还是Python 3呢？”答案并不会像我们想的那么明确。

Python现状如下：

1. 当前大部分生产级应用采用Python 2.7。
2. 当前Python 3已经可以用于生产级别的应用了。
3. Python 2.7将只会接受必要的安全更新，直到2020年[6]。
4. 现在”Python”这个名字指代Python 3和Python 2。

瞧，你也可以看到，这就是为什么不那么容易选择。

建议

我就直接说了：

- 对于新的项目，使用Python 3。
- 如果你是刚开始学习Python，熟悉下Python 2.7还是非常值得的，但还是推荐去学习Python 3，那会更加有用。
- 都学以下。它们都是“Python”。
- 目前已经构建的很多软件通常依赖于Python 2.7。
- 如果你正在开发新的开源Python库，最好同时支持Python 2和Python 3。对于想要被广泛使用的新库，如果你说只支持Python 3，那么这种政治声明会让很多用户疏远。 当然，同时支持两个版本不算多大个问题，接下来三年，这种情况会逐渐越来越少。

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

那么，用Python 3咯?

如果你要选择Python解释器，那么我推荐使用最新的Python 3.x, 因为每个新版本都会在标准库、安全性以及BUG修复上有所提升。

然而，如果你有很重要的原因只能使用Python 2，例如处理已经存在的Python 2代码、用到了Python 2独有的库、因为感觉更简单熟悉， 或者像我一样，极度喜欢Python 2，那么就使用Python 2。这也没什么坏处。

通过 Can I Use Python 3? 可以检查你依赖的软件是否会妨碍你采用Python 3。

进一步阅读

编写同时支持Python 2.6、2.7和Python 3 的代码是完全有可能的。至于难易程度，取决于你所编写软件的类型。如果你是新手，优先考虑其他更重要的方面更切实际。

实现

当人们讨论 *Python* 的时候，通常不仅仅意味着这门编程语言，同时也指CPython实现。*Python* 实际上是一个规范，该规范有多种实现方式。

CPython

CPython 是用C语言编写的一种参考实现，它会把Python代码编译成中间字节码供虚拟机来解释执行。CPython对Python的包和C扩展模块提供了最高程度的兼容。

如果你正在编写基于Python的开源代码，并且想让尽可能多的人受益，CPython可以说是最好的选择。如果使用的包依赖于C扩展，那么唯一的可选实现也只有CPython。

所有版本的Python语言都是C实现的，因为CPython是其参考实现。

PyPy

PyPy 是RPython实现的解释器，RPython是Python的子集，具有静态类型。这个解释器的特点是带有JIT编译器并且支持多种后端（C，CLI，JVM）。

PyPy的目标是在提高性能的同时，最大限度的保持与CPython参考实现的兼容性。

如果你想提高自己Python代码的性能，PyPy值得一试。在一套测试基准下，目前会 比CPython快5倍以上 。

PyPy支持Python 2.7。PyPy3 [1], 目前处于beta阶段，支持Python 3。

Jython

Jython 是Python的另一种实现，它会把Python代码编译为Java的字节码，然后被JVM（Java虚拟机）执行。另外，该实现也可以像使用Python模块一样来使用Java的类。

如果你需要与现有的Java代码进行交互，或者有其他原因需要在JVM上使用Python，Jython是最好的选择。

Jython目前支持到Python 2.7。 [2]

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

IronPython

IronPython 是.NET框架上的实现。可以同时使用Python和.NET的库，并且可以让.NET上的其他语言来调用所写的Python代码。

Python Tools for Visual Studio 把IronPython直接集成到了Visual Studio开发环境中，给Windows上的开发者提供了一个不错的选择。

IronPython支持Python 2.7。[3]

PythonNet

Python for .NET 作为一个包，为本地已安装的Python和.NET公共语言运行时（CLR)提供了无缝的集成。它采取与IronPython （见上文）相反的方法，与其说是竞争，不如说是互补。

通过结合Mono，pythonnet可以让安装在非Windows操作系统（比如OS X和Linux）上的Python与.NET框架进行互操作。除了IronPython外，它也是可以毫无冲突运行的。

Pythonnet支持范围从Python 2.6到Python 3.5。[4] [5]

- 如何正确地安装Python

1.2 正确地安装Python

其实，在你的系统中很可能已经内置安装好了Python。

如果是这样的话，不用额外的安装或者配置就能使用到Python。当然了，虽然话是这么说地，但还是强烈建议在真正使用Python做实际开发前，按照下面指南中描述的步骤安装Python的工具和库。特别是Setuptools、Pip和Virtualenv这几个工具一定要安装 - 它们会让你更方便地使用其他第三方的库。

安装指南

这部分内容涉及了 *Python* 、setuptools、pip以及virtualenv 的安装设置，为后续开发做准备。

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

在Mac OS X安装Python

注解： 查看 如何在OS X中安装Python 3.

最新的Mac OS X，Sierra **带有开箱即用的Python 2.7** 。

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

虽然说使用Python前不需要额外的安装或者配置，但是我强烈建议你在开始构建Python应用程序前，按照下一节描述的步骤安装工具和库。特别的，任何时候你都该安装Setuptools和pip，这样会让你更方便的使用其他第三方Python库。

OS X自带的Python版本对于学习来说绰绰有余，但是却不太适合用来开发。与 官方当前的版本 相比，自带的Python版本太旧了，没有达到生产环境稳定性的要求。

正确的方式

接下来我们安装一个真正的Python。

在安装Python前，需要先安装一个C编译器。通过运行 `xcode-select --install` 可以以最快方式安装Xcode命令行工具。你也可以从Mac应用商店下载完整版本的 Xcode ，或者是最小化的非官方版本 OSX-GCC-Installer 。

注解： 如果你已经安装了Xcode或者计划使用Homebrew，那就不要安装OSX-GCC-Install。同时安装二者时，软件会出现问题，并且很难诊断出原因。

注解： 如果是全新安装的XCode，还需要通过在终端运行 `xcode-select --install` 来添加命令行工具。

尽管OS X自带了大量的UNIX工具集，但是熟悉Linux系统的开发人员会发现缺少一个关键的组件：一个像样的包管理工具。Homebrew 填补了这块空白。

为了 安装Homebrew ，需要打开 `Terminal` 或者你最喜欢的OSX终端模拟器，然后执行如下命令：

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

这个脚本会在你安装之前说明它会对系统做出什么改变。一旦安装Homebrew，可以通过把下面一行内容添加到 `~/.profile` 中，来把Homebrew目录添加到环境变量 `PATH` 。

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

接下来，我们就可以安装Python 2.7：

```
$ brew install python
```

或者Python 3:

```
$ brew install python3
```

这会花费一两分钟。

Setuptools & Pip

Homebrew也会安装Setuptools和 `pip`。

Setuptools可以让你通过一个命令 `easy_install` 在网上下载和安装任何兼容的Python软件包。同时也可以很方便的利用它们在自己开发的Python软件里添加网络安装功能。

`pip` 是一个易于安装和管理Python包的工具，相比于 `easy_install` 更加推荐 `pip` 。它在 几个方面 要更加优于 `easy_install` ，并且维护良好。

虚拟环境

虚拟环境主要是通过为各自创建虚拟的Python环境，把不同项目所依赖的包分隔在各自独立的空间内。这样就能解决“项目X依赖版本1.x，但是项目Y需要版本4.x”的窘境，同时保持全局site-packages目录的干净和可管理性。

例如，你可以在一个需要Django 1.10的项目上进行开发工作，同时维护一个依赖Django 1.8的项目。

请参考文档 虚拟环境 来使用。也可以使用 *virtualenvwrapper* 来更简单的管理你的虚拟环境。

本章是 另外一篇文章 的修改合成版本，与原文使用同样的许可证。

在Windows上安装Python

首先，到官网下载Python 2.7的 最新版本 。如果你想确保安装的是最新版本，在 官网主页 上点击Downloads > Windows链接。

Windows的Python以MSI的格式提供，双击即可安装。MSI格式的文件允许Windows管理员使用标准工具来自动化安装。

按照设计，Python会安装到带有版本号的目录中，例如，Python 2.7会安装到 `C:\Python27\` ，这样可以在系统上安装多个版本而不引起冲突。当然，对于Python文件，只能有一个默认解释器。这种安装方式也不会自动修改环境变量 `PATH` ，这样你就可以控制运行哪个Python版本。

如果使用时，每次都要输入Python解释器的完整路径，显得太麻烦，所以最好把默认版本的Python目录添加到环境变量 `PATH` 。假设你的Python安装位置是 `C:\Python27\` ，把下述内容添加到 `PATH` :

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

在 `powershell` 中运行下列命令可以很容易做到这点:

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

当安装Python包时，一些新的命令会放置在上面的第二个目录（ `Scripts` ）中，所以把这个目录添加到环境变量中是很有用的。

虽然说使用Python前不需要额外的安装或者配置，但是我强烈建议你在开始构建Python应用程序前，按照下一节描述的步骤安装工具和库。特别的，任何时候你都该安装Setuptools和pip，这样会让你更方便的使用其他第三方Python库。

Setuptools + Pip

最至关重要的的第三方Python软件是Setuptools，该工具扩展了标准库中disutils的安装和打包功能。一旦安装好它们，你就可以使用一个简单的命令来下载、安装、卸载任何兼容的Python软件包。同时也可以很方便的利用它们在自己开发的Python软件里添加网络安装功能。

通过运行 ez_setup.py 这个脚本可以获得Windows上最新的Setuptools。

这时候就会新出现一个可以使用的命令：**easy_install**。不过这个命令已经被很多人认为废弃了，所以我们需要安装它的替代：**pip**。不像easy_install，Pip允许卸载安装好的包，并且维护也很活跃。

运行脚本 get-pip.py 可以安装pip。

虚拟环境

虚拟环境主要是通过为各自创建虚拟的Python环境，把不同项目所依赖的包分隔在各自独立的空间内。这样就能解决“项目X依赖版本1.x，但是项目Y需要版本4.x”的窘境，同时保持全局site-packages目录的干净和可管理性。

例如，你在一个需要Django 1.10的项目上进行开发，同时维护一个依赖Django 1.8的项目。

请参考文档 *Virtual Environments* 来使用。也可以使用 *virtualenvwrapper* 来更简单的管理你的虚拟环境。

本章是 另外一篇文章 的修改合成版本，与原文使用同样的许可证。

在Linux上安装Python

在最新版的CentOS、Fedora、Redhat Enterprise（RHEL）以及Ubuntu上已经 **搭载了开箱即用的Python 2.7**。

如果想查看自己安装的Python是什么版本，打开命令行运行如下命令：

```
$ python --version
```

一些老版本的RHEL和CentOS自带的Python 2.4对于现在的Python开发来说是无法接受的。幸运的是，存在 Extra Packages for Enterprise Linux ，这些包都基于Fedora的高质量包，而Fedora可以说是RHEL的试验场。这个仓库包含的Python 2.6包是定制的，可以与系统自带Python 2.4的包一一对应的安装。

虽然说使用Python前不需要额外的安装或者配置，但是我强烈建议你在开始构建Python应用程序前，按照下一节描述的步骤安装工具和库。特别的，任何时候你都该安装Setuptools和pip，这样会让你更方便的使用其他第三方Python库。

Setuptools & Pip

最至关重要的两个第三方Python包是 setuptools 和 pip。一旦安装好它们，你就可以使用一个简单的命令来下载、安装、卸载任何兼容的Python软件包。同时也可以很方便的利用它们在自己开发的Python软件里添加网络安装功能。

默认情况下，Python 2.7.9及之后（Python2系列）以及Python 3.4及之后的版本都已经包含了pip。

如果想知道pip是否已经安装，只需要打开命令行输入以下命令：

```
$ command -v pip
```

如果想自己安装pip，参考官方pip的安装教程 - 这样会自动安装setuptools的最新版本。

虚拟环境

虚拟环境主要是通过为各自创建虚拟的Python环境，把不同项目所依赖的包分隔在各自独立的空间内。这样就能解决“项目X依赖版本1.x，但是项目Y需要版本4.x”的窘境，同时保持全局site-packages目录的干净和可管理性。

例如，你可以在一个需要Django 1.10的项目上进行开发，同时维护一个依赖Django 1.8的项目。

请参考文档 虚拟环境 来使用。也可以使用 *virtualenvwrapper* 来更简单的管理你的虚拟环境。

本章是 另外一篇文章 的修改合成版本，与原文使用同样的许可证。

CHAPTER 2

写出优雅的代码

本章主要聚焦于编写Python代码的最佳实践。

组织好你的项目

这里的 “组织” 意思是，为了让你的项目能达到最佳目标而做出的抉择。我们需要考虑如何最大限度的利用Python的特性来创建简洁高效的代码。从实际中的角度来说，”组织” 就是为了让代码的逻辑和依赖足够清晰明了，类似于文件系统中文件和目录的组织那样一目了然。

哪些功能应该在哪些模块？项目中的数据流是如何进行的？哪些特性和功能应该被组织到一起或者分离开？通过回答这类问题就可以开始对你的项目有一些计划，换言之就是你对最终的产品形态有了一定的想法。

本节我们将详细了解Python的模块和导入系统，因为这两部分是组织好一个项目的核心元素。然后，我们会从多个方面讨论如何构建具有高扩展性和可靠性的代码。

仓库的组织

是的，这很重要！

正如在一个健康的开发周期中，代码风格、API设计以及自动化都是必不可少的一样，项目仓库的组织可以说是项目 架构 中最为关键的一部分。

当一个潜在的用户或者贡献者来到项目仓库的页面，他们会看到以下一些东西：

- 项目名
- 项目描述
- 一大堆的文件

只有当他们的滚动条到达目录的下方时，才会看到项目的README。

如果你的仓库里面是一大堆垃圾一样的文件或者各种杂乱嵌套的目录，即使你的文档再优秀，这些人也很有可能在阅读之前已经跑去其他地方了。

> 如果你以后想当总统，就不能穿的像个打工仔。

当然，第一印象不能代表一切。你和你的同伴可能会花费不计其数的时间在这个仓库上，最终对于项目里的各种犄角旮旯都能如数家珍。总之，仓库的设计依然是很重要的。

示例仓库

tl;dr（太长了，不读了）：这就是 Kenneth Reitz 的建议。

下面这个仓库 在GitHub 上可以找到。

```
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
```

我们来讨论下里面的一些细节。

真正的模块部分

位置	`./sample/` or `./sample.py`
用途	感兴趣的项目核心代码

这个模块是仓库的核心部分，这部分不应当被隐藏起来。

```
./sample/
```

如果你的模块仅仅包含一个单独的文件，你可以把这个文件直接放在仓库的根目录下。

```
./sample.py
```

你的这个库不应该放在一个模棱两可、叫做src的目录或者名为python的子目录。

许可证

位置	`./LICENSE`
用途	法律说明

这可以认为是除了代码本身之外仓库中最为重要的一部分。许可证的全文和版权申明都在这个文件中。

如果你不确定使用哪种许可证，到这里看看 choosealicense.com 。

当然了，你也可以不用包含许可证，不过这可能会潜在的阻碍许多人使用你的代码，毕竟怕出什么幺蛾子。

Setup.py

位置	`./setup.py`
用途	打包与分发管理

如果你的模块包在仓库的根目录下，很显然这个文件也应该在根目录下。

依赖说明文件

位置	`./requirements.txt`
用途	项目开发中的依赖

pip依赖文件 应该放置在仓库的根目录下。这个文件应该详细列出项目所依赖的库，包括测试、构建以及文档生成所用到的。

如果你的项目没有依赖，或者你更喜欢用 `setup.py` 来设置，那这个文件也可以不需要。

文档

位置	`./docs/`
用途	包的说明文档

毫无疑问，这部分别无他处。

测试套件

位置	`./test_sample.py` or `./tests`
用途	包的集成测试和单元测试

开始的时候，测试套件通常可能只是一个单独的文件：

```
./test_sample.py
```

一旦测试套件增多，就应该把这些测试移到一个目录中，就像下面这样：

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

很明显，这些测试模块需要从核心代码模块中导入来进行测试。具体可以通过以下几种方式：

- 假设待测试的包已经预先安装在site-packages中。
- 使用简单（但是 显式 ）的方式修改路径来正确的解析出包。

本人强烈推荐后一种方式。让一个开发人员通过运行 `setup.py develop` 来测试还在不断变化的代码，这种方式要求为每次改变后的代码部分设置隔离环境，太不友好了吧。

为了给测试提供独立的导入上下文，创建一个tests/context.py文件：

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

import sample
```

然后在这个独立测试模块中，按如下方式导入待测试模块：

```
from .context import sample
```

不管安装位置在哪里，这种方式总会按照预期工作。

一些人主张分发代码模块的同时一并分发测试代码 – 本人是反对的。这样通常会给你的用户增加复杂性，许多测试套件通常需要额外的依赖和运行时上下文。

Makefile

位置	./Makefile
用途	通用任务管理

如果你留意我的大部分项目或者Pocoo团队的任何一个项目，你会注意到都有一个Makefile文件。为啥？这些项目并不是C语言写的... 简而言之，make是一个极其有用的任务管理工具，可以简单的定义项目中常见的任务。（译者注：为毛不是SCons？）

Makefile示例：

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

test:
    py.test tests

.PHONY: init test
```

其他常见的管理脚本（例如 `manage.py` 或者 `fabfile.py` ）也应当在仓库的根目录下。

关于Django应用

自从Django 1.4发布以后，我在Django应用中注意到了一种新的趋势。由于Django自带的新模板系统，导致许多开发者把他们的项目仓库组织的很糟糕。

怎么个糟糕法？这么说吧，他们会进入新建的仓库目录，然后一如往常的执行如下命令：

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

其结果就是仓库结构看起来如下：

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

别再这么干了，伙计。

重复的路径会让你的工具和开发人员产生迷惑。不必要的嵌套与人无益（除非他们依然怀念庞大的SVN仓库）。

正确的做法是：

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

注意最后的那个“.“。

这种方式操作的结果如下：

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

代码的组织是关键

受益于Python处理导入和模块的方式，Python项目的组织相对容易很多。这里说的容易是指模块导入的概念很容易理解，并且不会受到太多的约束。因此，你可以更专注于项目中不同部分的交互，专注于纯架构上的任务。

换句话说，一个项目很容易组织也意味着可能会组织的很糟糕，毕竟组织起来可能会很随意。以下是一些糟糕组织方式的信号：

- 数量众多且杂乱的循环依赖：如果文件 `furn.py` 中的类Table和Chair需要导入文件 `workers.py` 中的类Carpenter（木匠），以便可以回答类似 `table.isdoneby()` 的问题，同时类Carpenter也需要导入Table和Chair来回答 `carpenter.whatdo()` 的问题，这时候就会产生循环依赖。这时候，你不得不采用一些奇技淫巧来解决这个问题，比如在方法或者函数中使用导入语句。
- 隐式耦合：每当对Table的实现进行微小改变时，就会由于破坏了Carpenter中的代码，进而导致不相关的测试套件中20来个测试无法运行，因此需要在修改时做小心翼翼的诊断工作。这也就意味着你对Carpenter代码中的Table需要做太多的假设，反之亦然。
- 大量使用全局变量或者上下文：Table和Carpenter依赖全局变量，而不是通过显示的互相传递 `(height, width, type, wood)` 来进行交互，而这些全局变量是可以被不同地方动态修改的。为了知道矩形的桌子为什么突然变成了正方形，你需要仔细检查每一个可能会接触到全局变量的地方，然后又发现远程模板代码也正在修改相关的上下文环境，使得桌子的外形更加难捉摸。
- 意大利面条式代码：所谓的意大利面条式代码就是占了几页的if语句和循环语句，同时还伴随着一坨没有良好分界、纯粹复制黏贴过来的代码。Python中通过缩进来组织代码的方式（这也是最有争议的一个特性）使得维护这样的代码很费力。不过好消息是，你不会见到很多这样的代码。
- 意大利饺子式代码：这种代码在Python中更常见，包含了数以百计相似的逻辑代码，这些代码片段都很小，通常是没有很好组织的类或者对象。就之前的题目而言，如果你从来没想到是否该用FurnitureTable、AssetTable或者Table甚至TableNew来完成手头的工作，那么你很可能已经在意大利饺子式的代码中遨游了。

模块

Python的模块可以说是目前已知的主要抽象层中的一个，也可能是最为自然的一个。抽象层允许把代码分割开来，与具体的数据和功能部分放在一起。

例如，项目中的一层可以处理与用户操作进行的对接部分，另一层处理低层次的数据操作。分离这两部分最自然的方式就是把接口性功能的代码放置在一个文件中，所有低层次操作放在另一个文件中。这种方式下，包含接口代码的文件需要导入包含低层次操作代码的文件。可以通过 `import` 和 `from ... import` 语句来实现。

一旦你使用了 *import* 语句，你就已经使用了模块。这些模块可能是类似 *os* 和 *sys* 的内置模块，环境中安装的第三方模块，抑或是项目中的内部模块。

为了与风格指南章节部分保持一致，模块名应该简短、小写，并且避免使用点号(.)或者问号(?)等特殊符号。所以，`my.spam.py` 这种文件名是应当避免的！如此命名会干扰到Python查找模块的方式。

在 *my.spam.py* 这种命名的情形中，Python会解释为去名叫 `my` 的目录中查找 `spam.py` 文件，显然这不是我们的初衷。这里有一个 示例 ，很好的说明了点号应当如何使用。

尽管你可以按照你的想法把你的模块命名为 `my_spam.py`，但是尽量少在模块名中使用下划线。

除了一些命名限制，没有其他特殊要求，Python文件就可以看作一个模块，但是，如果你想正确的使用模块的概念，避免一些问题，最好还是真正理解导入机制的原理。

具体来说，语句 `import modu` 会寻找同一目录下的文件 `modu.py` 作为调用者。如果同目录下没有找到，Python解释器会递归的在"path"中查找文件 `modu.py`，如果都没有找到，则引起ImportError的错误。

一旦文件 modu.py 找到，Python解释器就会在一个独立的作用域中执行这个模块。modu.py 中任何顶层的语句都会被执行，包括从其他模块中引入的那些。函数和类定义会存储到模块的字典里。

然后，模块中的变量、函数和类就可以在调用者中通过被导入模块的命名空间来使用。命名空间是Python编程中特别有用且功能强大的核心概念。

在许多编程语言中，有一个 include file 的指令来让预处理器把被包含文件中的代码拷贝到调用者中。Python中却并不是这样：被包含的代码有自己独立的模块命名空间，这意味着你通常可以不用太担心被包含的代码产生副作用，例如覆盖同名的函数等。

通过特殊的导入语法 from modu import * 可以模拟更加标准的行为。但这通常被认为是不好的习惯。**使用 import * 会使得代码很难读，不知道导入了些什么，并且使得依赖不那么封闭，用到没用到的都导入** 。

使用 from modu import func 这种方式可以很明确的导入需要的函数，并且放在模块的全局命名空间中。这样带来的危害会远远小于 import * 这种方式，因为可以显式的指明全局命名空间中导入的是什么，这种方式对于更简单的 import modu 而言，唯一的优势就是后面可以不用输入模块名，节省了一捏捏的输入成本。

不好的方式

```
[...]
from modu import *
[...]
x = sqrt(4)  # sqrt是modu的一部分？内置的？还是上面定义的？
```

较好的方式

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt是modu的一部分，当然了，前提是从import到这里中间没有重新定义过
```

最好的方式

```
import modu
[...]
x = modu.sqrt(4)  # 毫无疑问，sqrt就是modu的一部分
```

正如 代码风格 章节提到的，可读性是Python的主要特性之一。可读性意味着避免无用的重复文字和杂乱的东西，因此，Python中花费了不少努力来达到一定程度的简短。但是太简短会导致晦涩，所以简短应该在简洁和晦涩中找到一个平衡点。 modu.func 这种形式可以很直接的说明类或者函数来自哪里，对于不仅仅只包含一个文件的项目来说，可以极大的提高代码的可读性和可理解性。

包

Python提供了非常直接的包系统，就是简单的把模块机制扩展到了目录层面（译者注：模块是基于文件）。

任何包含 __init__.py 文件的目录都可以认为是一个Python包。包内的不同模块可以像普通模块那样导入，但是，__init__.py 文件比较特殊，这个文件主要用来把包内部的各种定义集中到一起。

目录 pack/ 中的文件 modu.py 可以通过语句 import pack.modu 来导入。该语句会去目录 pack 中寻找文件 __init__.py ，并执行这个文件中的顶层语句。然后再去寻找 pack/modu.py 文件，也执行其中所有的顶层语句。完成这些动作之后，modu.py 中的任何变量、函数或者类就都可以在pack.modu的命名空间中使用了。

一个常见的问题是在 __init__.py 中添加太多的代码。当项目的复杂程度逐渐增加时，目录结构的层次也会随之增加，子包以及子包的子包可能会在处于比较深的目录中。这种情况下，即使只是从子包的子包中导入很简单的一项，也可能需要在遍历目录树的过程中执行所有的 __init__.py 文件。

如果包中的模块和子包不需要共享代码时，通常会把 __init__.py 留空，这也是一种比较提倡的行为。

最后，对于深层次嵌套的包，有一种比较方便的语法 `import very.deep.module as mod` 。这可以让你使用 *mod* 来代替显示的指明 `very.deep.module` 。

面向对象编程

Pyhton有时候会被描述为是一种面向对象的语言。这可能稍微有点误导，这里需要澄清一下。

在Python中，任何东西都是一个对象，当然也就可以按照对象的方式来处理。比如，这意味着我们可以把函数当作第一类的对象来使用。Python中的函数、类、字符串，甚至是类型都是对象：就像任何其他对象一样，它们有类型，可以作为函数的参数传递，并且可以有自己的方法和属性。从这个意义上理解，Python确实是一种面向对象的语言。

然而，不像Java，Python并不强制必须使用面向对象来作为主要的编程范式。对于一些项目来说，不使用面向对象的方式完全可行，比如，可以不使用或者仅仅使用很少的类定义、类继承或者其他面向对象编程特有的机制。

此外，正如在 模块 一节中了解的那样，Python处理模块和命名空间的方式让开发人员可以很自然的确保封装性和抽象层次的分离，而这两者恰恰正是使用面向对象最常见的原因。因此，如果项目的业务模型不是必须使用面向对象来进行开发的话，Python程序员可以自由的选择不使用。

总会有一些理由避免不必要的面向对象。当我们想要把一些状态和功能粘合在一起的时候，自定义一个类是比较有用的方式。但是，正如在函数式编程讨论中指出的一样，问题恰好出在方程式中“状态”的部分。

在一些架构中，典型的例如Web应用，通常需要派生多个Python进程来同时相应外部的请求。这种情形下，在实例化的对象中保存某种状态（通俗点讲就是保存了所在上下文环境的一些静态信息）很容易引起并发问题或者竞争状态。有时候，从一个对象的状态初始化（通常是通过 `__init__()` 方法来完成）到通过其方法实际使用这个状态之间，上下文环境很可能已经发生变化，保留的状态也可能已经过期。例如，一个请求可能会载入某个条目到内存中，并且标记它已经被用户阅读过。与此同时，如果另一个请求需要删除这个条目，就会导致第一个请求载入的条目被这个请求删除掉，其结果就是我们标记阅读过一条不存在（已删除）的条目，这显然是不合理的。

由于这个以及其他一些问题，引发了使用无状态函数这种更好的编程范式的想法。

换种说法就是，建议在使用函数中，尽可能的少涉及隐式上下文和可能的副作用。函数的隐式上下文主要是指全局变量以及从函数内部访问的持久层对象。副作用就是指函数对隐式上下文做出了改变。如果函数保存或者删除了全局/持久层的数据，我们就说产生了副作用。

小心翼翼地把包含上下文和副作用的函数与包含逻辑处理的函数（纯函数）隔离可以得到如下好处：

- 纯函数是确定性的：如果输入是固定的，那么输出也一定是相同的。
- 如果需要重构或者优化时，纯函数更容易改变和替换。
- 纯函数更容易单元测试：很少需要设置复杂的上下文和清理事后的数据。
- 纯函数更容易操作、修饰以及传递。

总之，在一些架构中，纯函数在构建封闭块的时候比类和对象更有高效，因为没有上下文和副作用。

当然了，面向对象还是很有用的，甚至在很多情况下是必须的，比如开发图形桌面应用或者游戏，这种情况下，所操作的实体（窗口、按钮、头像、车辆等）本身在计算机内存中就会有很长的生命周期。

装饰器

Python语言提供了一个简单但是强大的语法，叫做“装饰器”。装饰器本身是一个函数或者类，可以用来包装（或者装饰）其他函数或方法。“被装饰”的函数或方法会替代原来“未装饰”的函数或方法。由于Python中函数是第一类对象，所以可以通过手动来实现函数的包装，但是使用@decorator语法的方式会显得更加清晰，因此也更加推荐这种用法。

```
def foo():
    # do something

def decorator(func):
    # 操作函数
    return func

foo = decorator(foo)  # 手工装饰

@decorator
def bar():
    # Do something
# bar()已被装饰
```

装饰器机制对于分离业务非常有用，可以避免外部不相关的逻辑“污染”函数或方法中的核心逻辑。其中一个很好的例子是 记忆表 或者缓存：对于一些运行代价较高的函数，你想把计算结果存在一个表里，以便后面需要的时候可以直接使用而不必重新计算，示例中的这一类功能可以很好的用装饰器来处理。很明显，这部分不能算是函数核心逻辑的一部分。

上下文管理器

上下文管理器是为一个操作提供额外上下文信息的对象。这个额外的信息采用如下形式来提供：在使用 `with` 语句初始化上下文的时候运行一个可调用对象（译者注：函数或者实现 `__call__` 的对象等），同时在执行完 `with` 块内部的所有代码后，再执行一个可调用对象。使用上下文管理器最为人熟知的例子就是打开一个文件，如下：

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

任何熟悉这种模式的人都知道，以这种方式调用 `open` 可以确保 `f` 的 `close` 方法在后面某个时间点会被自动调用。这可以减轻开发人员的记忆负担，同时也可以使得代码更容易阅读。

你自己可以采用两种方式来实现这种功能：使用类或者使用生成器。接下来让我们实现上面提及的功能，首先采用类的方式实现：

```
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()
```

这和普通Python对象没有什么区别，仅仅是多了两个额外的方法，这两个方法会被 `with` 语句使用。CustomOpen首先被实例化，然后它的 `__enter__` 方法会被调用，`__enter__` 返回的值会通过语句中的 `as f` 被赋值到 `f`。当 `with` 代码块中的内容被执行完毕时，`__exit__` 方法会被调用执行。

生成器实现的方式使用了Python内置的 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()
```

这种方式与上述类实现的结果完全一样，尽管简短了很多。首先 `custom_open` 逐句执行，直到到达 `yield` 语句处，然后把控制权交回给 `with` 语句， 然后会把 `yield` 产生的结果通过 `as f` 赋值到 f 上。`finally` 语句确保无论是否在 `with` 语句中产生异常，`close` 都能被调用。

由于这两种方式看起来没什么区别，所以我们应该遵循Python之禅来决定什么时候使用哪种方式。如果有大量的逻辑需要封装，那么类方式的实现可能更好。如果我们仅仅是执行一个简单的动作，那么函数的方式或许更好。

动态类型

Python是动态类型的，意味着变量没有固定的类型。事实上，Python中的变量与其他语言中的变量有着很大的不同，尤其是静态类型的语言。变量并不是写有某个值的计算机内存段，它们仅仅是指向对象的“标签”或者“名字”。因此，把一个变量设置为1，然后设置为“一个字符串”，再设置为一个函数是完全可以的。

Python的动态类型经常被认为是一个缺点，并且事实上的确会导致复杂性以及难以调试的代码。命名为“a”的变量可以被设置为很多不同的值，开发者或者维护人员需要在代码中跟踪这个名字，以便确保它不会被设置为一个完全不相关的对象。

一些准则有助于避免这类问题：

- 避免为不同的事物使用相同的变量名

糟糕的代码

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

好的代码

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

使用短小的函数或方法，有助于降低为无关事物使用相同命名的风险，毕竟作用域范围内代码量少了。

如果相关的事物有着不同的类型，最好分别使用不同的名字。

糟糕的代码

```
items = 'a b c d'  # 这是一个字符串...
items = items.split(' ')  # ...变身为列表
items = set(items)  # ...又变为了集合
```

重用名字并不能带来效率的提升：无论如何，赋值都会创建新的对象。然而，随着复杂性的增加，各个赋值语句会被很多行的代码分割开来，包括“if”分支和循环，这会使得要查明某个变量是什么类型变得更加困难。

在一些例如函数式编程的编码实践中，建议绝不要给一个变量重新赋值。在Java中，可以通过 *final* 关键字来做到禁止重新赋值。Python并没有 *final* 关键字，因为这会与它的哲学相违背。然而，避免给一个变量赋值超过一次是一个良好的习惯，同时，这也会有助于理解可变类型和不可变类型的概念。

可变与不可变类型

Python有两种内置类型或用户自定义类型。

可变类型就是那些允许在内容上直接修改的类型。典型的可变类型就是列表和字典：所有列表都有用于修改内容的方法，比如 `list.append()` 或者 `list.pop()`，可以直接在列表上进行修改。字典也是一样的。

不可变类型并不会提供修改自身内容的方法。比如，设置为整数6的变量x就没有"increment"方法。如果你想计算x+1，你不得不创建另外一个整数并命名。

```
my_list = [1, 2, 3]
my_list[0] = 4
print my_list  # [4, 2, 3] <- 列表本身已经改变

x = 6
x = x + 1  # 等号左边的x已经是另外一个对象，通过id(x)可以知道
```

两种类型在行为上的不同导致的结果就是，可变类型是不"固定的"，因此不能用作字典的键。

对于那些本质上会改变的事物应当使用合适的可变类型，对于那些本质上是固定的事物应当使用合适的不可变类型，这会使得代码的目的更加明确。

比如，与列表等价的不可变类型是元组，可以通过 `(1,2)` 来创建。这个元组包含一对不可以直接修改的值，因此可以用作字典的键。

Pyhton中的字符串是不可变类型，这可能会让初学者感到吃惊。这意味着，当要从一个字符串的各个组成部分构建字符串时，先把各个部分放到列表（可变类型）里，然后再用'join'方法粘合起来的方式会更加高效。然而，有一点需要注意的是，列表解析的方式比通过循环调用 `append()` 来构建列表更好也更快。

糟糕的代码

```
# 构建一个从0到19连接起来的字符串(比如："012..1819")
nums = ""
for n in range(20):
  nums += str(n)    # 低效且慢
print nums
```

较好的代码

```
# 构建一个从0到19连接起来的字符串(比如："012..1819")
nums = []
for n in range(20):
  nums.append(str(n))
print "".join(nums)  # 更加高效
```

优雅的代码

```
# 构建一个从0到19连接起来的字符串(比如："012..1819")
nums = [str(n) for n in range(20)]
print "".join(nums)
```

关于字符串最后需要提到的一点：使用 `join` 并不总是最好的选择。当需要从一定预设数量的字符串构建一个新的字符串时，使用加号操作符实际上更快，但是当类似之前提到的情况或者需要把字符串添加到一个已经存在的字符串上时，使用 `join()` 应该作为你的首选方式。

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

foobar = foo + bar  # 这种方式挺好
foo += 'ooo'  # 这种方式可就不好了，你应当采用如下方式：
foo = ''.join([foo, 'ooo'])
```

注解： 除了 `str.join()` 和 + 的方式之外，你也可以通过使用 % 格式化操作符来连接预设数量的字符串。然而， **PEP 3101**，不鼓励使用 % 操作符，而是更提倡使用 `str.format()` 方法。

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

foobar = '%s%s' % (foo, bar) # 马马虎虎了
foobar = '{0}{1}'.format(foo, bar) # 这样比较好
foobar = '{foo}{bar}'.format(foo=foo, bar=bar) # 再好不过了
```

第三方依赖

运行部件

进一步阅读

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

代码风格

如果询问一个Python开发者他最喜欢Python的哪一点，他们通常会说是其可读性。确实，高可读性是Python语言设计的核心准则之一，主要是基于这样一个事实：阅读代码要远多于编写代码。

Python代码之所以容易阅读和理解，原因之一就是它相对完整的编码风格指南以及“Pythonic”的惯用方式。

此外，当一个富有经验的Python开发者（一个Pythonista）指出一部分代码不够“Pythonic”的时，通常意味着这部分代码没有遵循通用的风格指南，并且没有按照最佳方式（即：最具有可读性）来进行缩进处理。

一些边际情况并没有统一的最佳实践方式来进行Python代码缩进，但是这种情况还是比较罕见的。

常规概念

显式的代码

尽管在Python中可以使用任何的黑魔法，但是更提倡显式直白的方式。

坏的代码风格

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

好的代码风格

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

在上述好风格的代码中，x和y可以从调用者那里显式的获取，然后返回一个显式的字典。使用这个函数的开发人员可以通过阅读代码的首行和尾行清楚的知道它是干什么的，而坏风格的代码则无法做到这点。

一行一语句

列表解析这种组合语句是被允许和鼓励的，这主要是由于其简洁和富有表现力，尽管如此，把两个不太关联的语句放在同一行却不是一种好的实践方式。

坏的代码风格

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

if x == 1: print 'one'

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

好的代码风格

```
print 'one'
print 'two'

if x == 1:
    print 'one'

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

函数的参数

参数可以通过四种方式来传递给函数。

1. **位置参数** 是强制的且没有默认值。这是最简单的参数形式，这种方式可以被用在很少参数即可表达完整意义的函数中，并且这些参数顺序是很自然的。举个例子，在函数 `send(message, recipient)` 或者 `point(x,y)` 中，函数的使用者可以毫不费力的记住这两个函数需要两个参数，以及参数的顺序是什么。

   在这两个示例中，可以通过使用参数名来调用函数，如果采用这种方式，参数的顺序是可以交换的，调用方式为 `send(recipient='World', message='Hello')` 以及 `point(y=2, x=1)`，但是与直接调用 `send('Hello', 'World')` 和 `point(1, 2)` 相比，这会降低可读性，同时也造成了不必要的冗长。

2. **关键字参数** 不是强制的且可以有默认值。它们通常被用于发送给函数的可选参数。当函数有多于两个或三个位置参数时，函数签名会变得相对难记，这时，使用带有默认值的关键字参数会有帮助的多。例如，更加完整的 `send` 函数可能被定义为 `send(message, to, cc=None, bcc=None)`。这里的 `cc` 和 `bcc` 是可选的，并且，在没有被传递其他值时，会被赋值为None。

   在Python中，调用带有关键字参数的函数有多种方式，比如，可以按照函数定义时参数的顺序来调用，这时不需要显式的命名参数，就像 `send('Hello', 'World', 'Cthulhu', 'God')` 中一样，发送秘密抄送给上帝。同样，也可以用命名参数的方式来按其他顺序调用，

就像 `send('Hello again', 'World', bcc='God', cc='Cthulhu')` 。除非有很重要的原因，否则上述两种方式最好避免使用，而应该按照最接近函数定义的语法方式来调用: `send('Hello', 'World', cc='Cthulhu', bcc='God')` 。

作为附注，参见 YAGNI 准则，通常来说，移除那些似乎永远用不到但为了"以防万一"而添加的可选参数（以及它在函数内的逻辑部分），要比需要时再添加新的可选参数以及其逻辑要困难的多。（译者注：也就是说，如无必要，不必预留不太可能用到的可选参数）

3. **任意参数列表** 是传递给函数参数的第三种方式。如果函数的意图可以通过一个包含有数目可扩展的位置参数的函数签名表达出来，那么，可以定义一个使用了 `*args` 参数的函数。在函数体内， `args` 会是一个剩余位置参数组成的元组。例如， 可以使用每个接收者作为参数来调用 `send(message, *args)` : `send('Hello', 'God', 'Mom', 'Cthulhu')` ，在函数体内 `args` 等同于 `('God', 'Mom', 'Cthulhu')` 。

   然而，这种构造有一些缺点，使用时应当谨慎。如果一个函数接收一组具有相同属性的参数，那么把函数定义成接收列表形式或者任意序列形式参数的方式会更加清晰。此例而言，如果 `send` 有多个接收者，最好显式的把它定义为： `send(message, recipients)` ，并且以 `send('Hello', ['God', 'Mom', 'Cthulhu'])` 方式调用。这样，函数使用者可以以预先定义好的列表形式来操作一组接收者，同时也打开了传递任何序列的可能性，包括无法解包为其他序列的迭代器。

4. **关键字参数字典** 是最后一种函数传递参数的方式。如果函数需要一系列未确定的命名参数，可以使用 `**kwargs` 构造。在函数体内， `kwargs` 是一个由所有尚未被函数签名中关键字参数捕获的其他命名参数的字典。

   与 任意数目参数列表 中一样，也必须谨慎使用这种方式，原因也是相似的：这种强力的技术应该在必要的时候才使用，如果存在更简单更清晰的方式即可满足函数的意图，那么应该避免使用 关键字参数字典 这种方式。

哪些参数作为位置参数，哪些参数作为可选的关键字参数，是否使用传递任意数目参数的高级技术，这都是由编写函数的开发者来决定的。如果明智的采用上述建议，完全有可能愉快的写出符合下列条件的函数:

- 易于阅读（函数名和参数无需过多解释）
- 易于修改（添加新的关键字参数不会破坏代码的其他部分）

避免魔法方法

作为黑客的强力的工具，Python自带了非常丰富的钩子和工具，允许你完成几乎任何奇技淫巧的事情。例如，它可以完成以下任何一件事:

- 改变对象的创建和初始化方式
- 改变Python解释器导入模块的方式
- 在Python中嵌入C代码（如果需要的话，建议这么做）

然而，所有这些选择都有许多缺点，所以使用最为直接的方式来达到你的目的总会更好。最为主要的缺点是使用这些构造方式严重影响了可读性。许多代码分析工具，例如pylint或者pyflakes将无法解析这些"魔幻的"代码。

我们认为Python开发者应该了解这些几乎无限的可能性，因为这会给你灌输自信，让你觉得没有不可逾越的难题。然而，知道怎么使用以及明确何时 **不** 去使用它们却非常重要。

就像功夫大师一样，一个Pythonista知道如何用一根指头杀人，然而却永远不会这么做。

我们都是负责的用户

如上所见，Python允许很多技巧，但是其中一些具有潜在的危险性。一个比较好的例子是客户端代码可以重写对象的属性和方法：在Python中，没有“private”关键字。这是Python的哲学，与像Java这样具有高度防御性的语言不同，高防御性语言会提供许多机制来阻止任何的误用，而Python会通过表明：我们都是负责的用户来达到这点。

这并不意味着，属性不能被认为是私有的，抑或Python无法进行合适的封装。相反，Python并不依赖于开发者在自身代码和其他人的代码之间竖立坚固的墙来达到隔离，Python社区更倾向于依赖一系列的惯例来表明这些元素不应当被直接访问。

对于私有属性，主要的惯例和实现细节是对所有“内部的元素”使用下划线前缀。如果客户端代码破坏这个规则，并且访问这些标记的元素，遇到的任何不正确行为或者问题都应当由客户端代码负责。

我们鼓励慷慨的使用这些惯例：任何不计划被客户端代码使用的方法或者属性应当使用下划线作为前缀。这样可以确保更好的责任分离以及对已有代码更容易的修改；把私有属性公有化总是可行的，反之，把公有属性私有化则困难的多。

返回值

随着函数复杂性的增长，在函数体内使用多个返回语句变得很常见。然而，为了保持函数意图明确以及维持足以接受的可读性，更倾向于避免从函数体的多个出口点返回有意义的值。

在一个函数中返回值主要有两种情况：一种是函数正常处理完毕返回结果，另一种是返回错误情况，以便说明由于错误的输入参数或者其他原因，进而导致函数无法完成计算或任务。

如果在第二种情况下你不希望抛出异常，那么应当返回一个None或者False值来表明函数无法正常处理。这种情况下，最好在检测到不正确的上下文时尽早返回。这样有助于函数结构的扁平：返回语句（由于错误而返回）之后的代码可以认为是满足后续计算函数结果的情形。函数中往往会有多个这样的返回语句（由于错误而返回）。

然而，当一个函数在正常路径上有多个主要的退出点时，会导致难以调试返回结果，所以如果可能，应当保留单个退出点。这将有助于提取一些公共的代码路径，并且如果有多个退出点也说明函数很有可能需要重构。

```
def complex_function(a, b, c):
    if not a:
        return None  # 抛出异常可能会更好
    if not b:
        return None  # 抛出异常可能会更好
    # 尝试从a, b和c中计算x的复杂代码
    # 如果成功，暂时先不返回x
    if not x:
        # 计算x的其他方式
    return x  # 返回值x有单一的退出点有助于代码的维护
```

惯用语法

编程习惯，简而言之就是写代码的 方式 。在 c2 和 Stack Overflow 上有着对编程习惯广泛的讨论。

惯用的Python代码通常可以称为 *Pythonic* 的代码。

尽管通常有一种（当然，最好也只有一种）显而易见的方式来写惯用代码，但是对于Python初学者来说，如何写出符合语言习惯的Python代码却并不那么明显。所以，好的编程习惯必须主动学习才能获得。

一些通用的Python惯用语法如下：

解包

如果你知道列表或者元组的长度，你可以通过解包来给其中的元素分配名字。例如， `enumerate()` 会为列表中的元素生成一个二元组：

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

你也可以使用这种方式来交换变量：

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

嵌套的部分也可以解包：

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

在Python 3中，通过 **PEP 3132** 引入了一种新方法来扩展解包方式:

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

创建可忽略的变量

如果你需要把某值赋给变量（例如，在 解包 中），但是又不会真正用到这个变量，那么可以使用 __ ：

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

注解： 许多Python风格指南建议使用单个下滑线 “_” 来处理那些用不到的变量，而不是这里建议的双下划线 “__” 。这种方式的问题在于 “_” 通常会被用作 `gettext()` 函数的别名，同时，在交互式环境中，单下划线往往保存着最后一次操作的结果值。而双下划线与单下划线一样清晰方便，且消除了这两种情形下意外干扰的风险。

创建长度为N的且由相同元素组成的列表

使用Python列表的 * 操作符：

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

创建长度为N且元素为列表的列表

由于列表是可变的， * 操作符（如上）会创建一个包含有N个指向 同一 列表引用的列表，这种方式并不是我们想要的。这种情况下，我们使用列表解析：

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

注意：在Python 3中要使用range()代替xrange()

从列表创建字符串

创建字符串的通用惯例是在空字符串上调用方法 `str.join()` 。

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

这种方式会给变量 *word* 赋值为“spam”。这种惯用方式适用于列表和元组。

在聚合集中搜索元素

有时候我们需要在聚合集中进行查找。这里我们来看看两种结构的查找方式：列表和集合。

代码示例：

```
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
```

尽管两个函数看起来完全一样，但是由于 *look_set* 利用了Python集合属于哈希表的特性，二者之间的性能差异极大。为了确定一个元素是否在列表中，Python不得不遍历每一个元素，直到找到匹配的元素为止。这是很耗时的操作，尤其是列表很长的时候。另一方面，在集合中，元素的哈希值会直接告诉Python去哪里查找匹配的元素。所以即使集合再大，也可以很快的完成查找。字典中的查找方式也类似集合。更多信息请参见 StackOverflow 。如果想知道各种常用操作在这些数据结构上耗费时间的详细信息，请参见 此页 。

由于性能上的差异，以下情形使用集合或者字典来代替列表是个不错的主意：

- 聚合集包含有大量的元素
- 需要不断重复的在聚合集中搜索元素
- 没有重复的元素

对于一些小的聚合集，或者是不需要进行频繁搜索的聚合集，创建哈希表所花费的时间和内存，往往会比由于搜索速度提升而节省出的时间更多。

Python之禅

因 **PEP 20** 为人熟知，这是Python设计的指导准则。翻译 在此 。

```
>>> import this
Python之禅, by Tim Peters

优美胜于丑陋，明晰胜于隐晦。
简单胜于复杂，复杂胜于繁芜。
扁平胜于嵌套，稀疏胜于密集。
可读性很重要。
虽然实用性比纯粹性更重要，
但特例并不足以把规则破坏掉。

错误状态永远不要忽略，
除非你明确地保持沉默，
直面多义，永不臆断。
```

```
最佳的途径只有一条，然而他并非显而易见--------谁叫你不是荷兰人?

置之不理或许会比慌忙应对要好，
然而现在动手远比束手无策更好。

难以解读的实现不会是个好主意，
容易解读的或许才是。

名字空间就是个顶呱呱好的主意。

让我们想出更多的好主意！
```

这里有一些符合Python风格的例子，参见 这些幻灯片来自Python用户组 。

PEP 8

PEP 8 是Python事实上的代码风格指南。pep8.org 上有一份高质量且易读的PEP 8版本。

强烈建议阅读这份指南。整个Python社区都尽最大努力遵守这份文档中提及的指导。一些项目可能会随着时间推移逐渐偏离其指导，而另外一些则会 改善其中的建议 。 总之，确保你的代码遵循PEP 8通常来说是个不错的主意，并且与其他开发者合作时，这也有助于代码风格的统一。有一个命令行工具 pep8 ，可以帮助你检查代码是否符合规范。在终端执行下面的命令来安装：

```
$ pip install pep8
```

然后在需要检查的文件上运行这个命令，就可以得到检测报告：

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

工具 autopep8 可以自动把代码重新格式化到符合PEP 8风格。安装方式如下：

```
$ pip install autopep8
```

可以用这个工具来直接格式化并修改文件：

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

如果除去 `--in-place` 标志，它会把格式化后的代码直接输出到终端，以便查看。 `--aggressive` 标志会进行更大的修改，可以通过多次使用这个标志来达到更好的格式化效果。

约定

你应当按照本节的约定，以便你的代码更容易阅读。

检查变量是否等于常量

对于一个值，你并不需要显示地把它与True、None或者0进行比较 - 只需要把它放到if语句中即可。参见 Truth Value Testing 了解哪些值可以认为是false。

坏的代码风格:

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

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

好的代码风格:

```
# 只需检查值即可
if attr:
    print 'attr is truthy!'

# 或者检查值的相反情况
if not attr:
    print 'attr is falsey!'

# 又或者，由于None可以被认为是false，可以显示的检查一下
if attr is None:
    print 'attr is None!'
```

访问字典元素

不要使用 `dict.has_key()` 方法。取而代之，使用 `x in d` 的语法形式或者给 `dict.get()` 传递一个默认值。

坏的代码风格:

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

好的代码风格:

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

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

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

操作列表的简便方式

列表解析 提供了一种强大简洁的方式来操作列表。此外，`map()` 和 `filter()` 函数使用了不同却更加简洁的语法。

坏的代码风格:

```
# 过滤大于4的元素
a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)
```

好的代码风格:

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

坏的代码风格:

```
# 对每个列表元素加3
a = [3, 4, 5]
for i in range(len(a)):
    a[i] += 3
```

好的代码风格:

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

使用 `enumerate()` 来获取元素在列表中的位置。

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

`enumerate()` 函数相对于手动计数有着更好的可读性，而且有助于迭代器的优化。

读取文件

使用 `with open` 语法来读取文件，这种方式会自动关闭文件。

坏的代码风格:

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

好的代码风格:

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

`with` 语句是一种更好的选择，因为这种方式会确保文件的关闭，即使在 `with` 代码块中抛出异常也能正确处理。

延续代码行

当代码的逻辑行长度超过限制时，需要把代码分割成逻辑上关联的几行。如果一行代码的最后一个字符是反斜杠，那么Python解释器会自动把后续的行连接起来。在一些情况下，这种做法很有用，但是通常应当避免这种方式，因为这种处理方式比较脆弱：行末尾反斜杠之后的空白会破坏代码并且导致一些无法预期的结果。

更好的解决方案是使用括号。如果一行开头有左括号，但是行末却没有相应的右括号，Python解释器会把下一行连接起来，直到遇到关闭的右括号。对于大括号和方括号也有类似的行为。

坏的代码风格:

```
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
```

好的代码风格:

```
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)
```

然而，多半情况下，当不得不分割一行很长的代码时，往往预示着你同时尝试完成的功能太多，这有可能会妨碍可读性。

阅读优秀代码

Python设计背后的核心理念之一就是创建高可读性的代码。这种设计背后的动机很简单：Python程序员首要做的事就是阅读代码。

成为优秀Python开发者的秘诀之一就是读代码，了解代码，然后深入理解优秀的代码。

译者注：下面是Youtube的视频，请自备梯子。

优秀的代码都会遵循 代码风格 中罗列的指导规范，并且会尽可能向其读者表达出清晰简洁的意图。

下面罗列一些推荐阅读的Python项目。这里的每一个项目都是Python编码的模范之作。

- Howdoi 是一个搜索工具，采用Python开发。
- Flask 是基于Werkzeug和Jinja2的Python微框架。其目标：为实现构想而进行快速开发。
- Diamond 是一个Python守护程序，可以搜集监测数据并发送到 Graphite 或者其他后端。目前可以搜集包括CPU、内存、网络、IO设备、负载以及磁盘在内的相关数据。另外，它还提供了API来实现定制化的搜集器，可以从几乎任何来源搜集数据。
- Werkzeug 开始只是供WSGI应用使用的一些简单的工具集合，后面逐渐发展成为高级WSGI工具模块中的一员。其中包括了强大的调试器、特性齐全的请求/响应对象、处理实体标签的HTTP工具、缓存控制头、HTTP日期、Cookie处理、文件上传、强大的URL路由系统以及一大堆社区贡献的其他模块。
- Requests 是一个基于Apache2许可证的HTTP库，采用Python开发，非常人性化。

- Tablib 是一个与格式无关、实现数据集表格化输出的库，采用Python开发。

待处理

增加上述每个项目里代码的典型示例。解释为什么这是优秀的代码。可以使用一些较为复杂的示例。

待处理

讲述一些可以快速确定数据结构、算法并确定代码实现什么功能的技术。

文档

无论是项目文档还是代码文档，可读性都是Python开发人员重点关注的一方面。遵守一些简单的最佳实践可以为你和他人节省出一大堆时间来。

项目文档

`README` 文件在项目根目录下，主要给出项目维护者以及用户的常规信息。此文件最好是使用纯文本或者一些容易阅读的标记文本来编写，例如使用 *reStructuredText* 或者Markdown。文件中应当包含几行内容来说明本项目或者本库主要用来干什么（要假设用户对此项目一无所知）、软件源代码的URL以及一些基本的信用信息。这个文件可以说是代码阅读人员的主要入口。

`INSTALL` 文件在Python项目中不是那么的必要。安装指导通常只是一个命令，比如 `pip install module` 或者 `python setup.py install`，可以直接添加到 `README` 文件中。

`LICENSE` 文件应该总是存在，并且需要指明软件在什么许可证下对公众可用。

`TODO` 文件或者是 `README` 中的 `TODO` 章节应当列出代码的开发计划。

`CHANGELOG` 文件或者 `README` 中的 `CHNANGELOG` 章节应当为最新版本的代码基变更作出一个简短的说明。

项目发布

依项目不同，通常你的文档可能会由以下全部或者部分的内容构成：

- 简介 主要是对该项目产品可以干什么作出一个非常简短的说明，可以使用一个或者两个极其简单的例子。这可以说是对你项目进行一个30秒的推销。
- 指南 应当更加详细地展示一些初级的案例。读者可以一步一步根据案例来搭建一个可工作的原型。
- *API*参考 通常直接由代码来生成（参见 *docstrings*）。其中会列出所有公开可用的接口、参数以及返回值。
- 开发文档 主要是为潜在的贡献者人员提供。其中会包括代码约定以及项目的常规设计策略等。

Sphinx

Sphinx 无疑是最为流行的Python文档工具。**赶紧去使用吧。** 它会把 *reStructuredText* 标记语言转换为各种格式的输出，包括HTML、LaTeX（用于可打印的PDF）、man手册以及普通文本。

网上还有一个 **非常好用** 且 **免费** 的 Sphinx 文档托管网站: Read The Docs 。赶紧去使用吧。通过配置它的提交钩子到你的源码仓库，可以实现自动化构建你的文档。

运行 Sphinx 时，会自动导入你的代码，并利用Python的内省机制提取出代码中的函数、方法以及类签名。同时还会提取出相应的文档字符串，并编译成结构良好又易于阅读的项目文档。

注解： Sphinx以由API生成文档而闻名，但是对于常规项目文档的生成也可以完成的很好。本文档就是使用 Sphinx 构建并托管于 Read The Docs 上。

---

reStructuredText

大部分的Python文档使用 reStructuredText 来编写。这种标记语言就像是内建了各种可选扩展的Markdown。

reStructuredText Primer 和 reStructuredText Quick Reference 可以帮助你熟悉其语法。

译者注: 网上有中文教程，请自行搜索。

代码文档建议

注释可以阐明代码，它们主要用来让代码更加容易理解。在Python中，注释以 # 开头。 在Python中， 文档字符串*(docstrings)* 描述了模块、类以及函数:

```
def square_and_rooter(x):
    """返回自身乘以自身后的平方根"""
    ...
```

通常可以参照 **PEP 8#comments** （Python风格指南）中的注释那一节。关于文档字符串的更多信息可以在 **PEP 0257#specification** （文档字符串约定指南）中找到。

注释代码片段

不要使用三引号字符串来注释代码 。这种方式并不是一个好的实践，因为类似grep这种面向行操作的命令行工具，是无法知晓那部分代码已失效的。更好的方式是确保正确的缩进，并在每个注释行前添加 # 。你使用的编辑器说不定可以很容易的完成这种功能，所以学习下如何注释/取消注释是很值得的。

文档字符串与魔法

一些工具会使用文档字符串来实现一些不仅限于文档的行为，比如单元测试的逻辑。这看上去很不错，但是你不会因为“这里就是这么做的”而永远不出错。

Sphinx 会把你的文档字符串解析为reStructuredText，然后渲染成HTML，这就可以很方便的把示例代码嵌入到文档项目中。

另外, Doctest 会读取所有格式为Python命令行输出样式（以”>>>”为前缀）的文档字符串，然后执行这部分文档内容，检测命令的结果是否匹配紧接着的下一行内容。开发人员可以利用示例代码和函数使用说明一起来注释源码，顺便还可以确保代码被测试通过。

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

```
    """
    return a * b
```

文档字符串与块注释

二者并非不可交换。对于一个函数或者类，开头的注释块是开发人员的笔记说明。文档字符串则描述了函数或者类进行的 操作 。

```
# 这个函数会由于某些原因降低程序的运行速度
def square_and_rooter(x):
    """返回自身乘以自身后的平方根"""
    ...
```

与块注释不同，文档字符串内建于Python语言自身。这意味着在运行时你可以使用Python强大的内省能力来访问文档字符串，相比而言，注释则会被优化掉。几乎每一个Python对象都可以从 *__doc__* 属性或者内建的 *help()* 函数来访问文档字符串。

块注释通常用于解释一段代码是做什么的，或者阐述一个算法，而文档字符串更倾向于向别人解释你的代码中的某个函数如何使用以及一个函数、类或模块的主要目的是什么。

编写文档字符串

根据所写函数、方法或类的复杂性，有时候单行文档字符串非常适用。以下是一个非常鲜明的例子:

```
def add(a, b):
    """把两个数字相加，返回结果"""
    return a + b
```

文档字符串应该以一种非常易懂的方式来描述函数。对于一些不重要的函数或者类，简单的把函数签名（比如： *add(a, b) -> result* ）嵌入到文档字符串完全没必要。因为如果需要的话，使用Python的 *inspect* 模块可以很容易找到这些信息，而且通过阅读代码也可以很容易明白。

然而，在更大更复杂的项目中，最好还是对一个函数给出足够多的信息：它做了什么、有可能抛出什么异常、返回什么或者一些参数的相关细节。

Numpy项目使用了一种流行的文档风格来给出代码更详细的信息，称为 Numpy风格 的文档字符串。尽管这种注释风格会比之前的示例占用更多行，但是也让开发人员可以为一个方法、函数或类提供更多的信息:

```
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
```

插件 sphinx.ext.napoleon 可以让Sphinx解析这种风格的文档字符串，方便你把Numpy风格的文档字符串包含到项目中。

最后，采用什么风格编写文档字符串并不重要，它们的目的都是为那些需要阅读或者修改你代码的人而服务的。只要这些文档是正确的、可以理解的并且你得到了想要知道的，那么赋予它的使命就算完成了。

如果还想对文档字符进一步的了解，参考 **PEP 257**

其他工具

你可能会在其他地方看到这些工具。参见 *Sphinx* 。

Pycco Pycco是一个“文学编程风格的文档生成器”，是node.js里 Docco 的移植。它可以把代码转化成代码与文档并排展现的HTML格式。

Ronn Ronn可以构建Unix的man手册。它可以把人类可读的文本转换为用于终端显示的roff格式以及用于Web的HTML格式。

Epydoc Epydoc已经停止开发。使用 *Sphinx* 替代吧。

MkDocs MkDocs是一个快速简单的静态网站生成器，致力于使用Markdown来构建项目文档。

测试你的代码

对自己的代码进行测试是非常重要的一步。

同时编写测试代码和实际运行的代码是一个非常好的习惯。合理利用这种方法可以帮助你更加精确的定位代码功能，并构建出一个更加解耦的架构。

测试中一些通用的准则：

- 一个测试单元应当聚焦于一个很小的功能并证明其实现的正确性。
- 每一个测试单元必须保持完全独立。保证既可以单独运行，又可以集成在测试套件中运行，而不必考虑其调用的顺序。这个规则暗含这样的意思：每个测试载入的数据集必须是全新的，并且完成测试后需要清理这些数据集。这通常通过方法 `setUp()` 和 `tearDown()` 来实现。
- 尽可能的使得测试集可以快速运行。如果单单一个测试就需要运行几毫秒的话，不仅会降低开发速度，也可能无法按照需要的那样尽可能频繁的运行测试。在某些情况下，测试集由于工作在复杂的数据结构上，且每次都要载入这些数据结构，所以测试无法快速运行。这时，可以把这种比较重量级的测试集放在单独的测试套件中，该测试套件通过计划任务来运行，而其他测试集则可以尽可能频繁的运行。
- 学习使用工具，并学习如何运行一个单独的测试或者测试用例。这样，当在模块中开发一个函数时，可以频繁的运行这个函数的测试，理想的方式是每次保存代码时自动运行对应的测试。
- 在每次进行编码任务前运行全套的测试套件，完成此次编码任务后再次运行。这会让你更确信自己的编码没有对代码的其他部分造成任何破坏。
- 通过实现钩子，把代码推送到共享仓库之前自动运行所有测试集是一个好的主意。
- 如果你正在进行开发中，并且由于其他原因不得不打断目前的开发工作，那么，对接下来要开发的部分编写一个损坏的单元测试是个不错的主意。当你回来继续工作时，你可以有个类似指针一样的东西告诉你目前工作在哪个地方，这样就可以快速回归开发状态。

- 调试代码的第一步：编写一个新的测试来精确的暴露出BUG。尽管不可能总这样做，但是，那些捕获BUG的测试集可以说是项目中最具有价值的一部分。
- 给测试函数命名一个长的具有描述性的名字。这里的代码风格与实际运行的代码略微有一些不同，对于实际运行的项目代码，我们更倾向于短命名。原因在于，测试函数从不会被显示的调用。在实际运行的代码中 `square()` 或者 `sqr()` 都是可以的，但是在测试代码中，你应该命名为 `test_square_of_number_2()` 、 `test_square_negative_number()` 。这些函数名会在测试失败时显示，所以应当尽可能的具有描述性。
- 当出现错误或者不得不进行改变时，如果有一个好的测试集，你和其他维护者就可以大量的依赖测试套件来修复问题所在，或者修改给定的行为。因此，相比于实际运行的代码，测试代码被阅读的次数至少会与其一样多，甚至更多。这种情况下，目的不清晰的单元测试用处并不大。
- 测试代码的另外一个用途是作为新开发人员的入门介绍。当有人不得不在代码基上进行工作时，运行并阅读相关的测试代码通常是他们开始的最好方式。他们可以发现热点所在、哪些地方是难点以及一些边边角角的情形。如果他们想添加某个功能，第一步要做的就是添加一个测试，通过这种方法，确保新功能不再是一个没有嵌入到接口中的工作路径。

基础知识

单元测试

`unittest` 是Python标准库内置的测试模块。如果使用过JUnit/nUint/CppUnit系列工具中的任何一个，你会发现这个模块的API非常熟悉。

可以通过实现 `unittest.TestCase` 的子类来完成创建测试用例。

```
import unittest

def fun(x):
    return x + 1

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

Python 2.7 标准库中的单元测试模块 包含自己的测试发现机制。

文档测试

`doctest` 模块会在文档字符串中搜索看起来像交互式Python会话的文本片段，然后执行这些会话，以确认是否可以如展示的那样工作。

文档测试与单元测试有着不同的使用情形：它们通常很少详细的描述，不会捕捉特殊的情况或者不明显的回归Bug。它们的主要用途是作为表述性文档来描述模块及其组成的主要部分。然而，每次完整运行测试套件时，文档测试也应当自动运行。

函数中简单的文档测试：

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

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

```
    return x * x

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

当以 python module.py 方式在命令行中运行这个模块时，文档测试就会运行，并且，如果没有按照文档字符串中描述的行为执行，会发出提示。

工具

py.test

py.test是Python标准库中unittest模块的可选替代。

```
$ pip install pytest
```

尽管是一个特性齐全、可扩展的测试工具，但是它具有简单的语法。创建一个测试套件如同编写一个只包含有几个函数的模块一样简单：

```
# test_sample.py的内容
def func(x):
    return x + 1

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

然后运行 *py.test* 命令

```
$ 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 =========================
```

与unittest模块相比，同样的功能，py.test需要更少的工作。

Nose

nose扩展了unittest来使得测试更加容易。

```
$ pip install nose
```

nose提供了自动发现测试集的功能，避免了人工创建测试套件的麻烦。同时也提供了大量的插件来支持兼容xUnit的测试输出、覆盖率报告以及测验选择等特性。

nose

tox

tox是一个管理自动测试所需环境以及配置多解释器测试的工具。

```
$ pip install tox
```

tox使用简单的ini格式配置文件，以便允许你配置复杂的多参数测试模型。

tox

Unittest2

unittest2是Python 2.7中unittest模块的移植，该模块拥有增强的API以及更好的断言，超越了之前Python中对应的模块。

如果你使用Python 2.6或者更低的版本，可以通过pip来安装：

```
$ pip install unittest2
```

可以采用unittest名自来引入该模块，这样，将来移植代码到模块的新版本时会更加容易。

```
import unittest2 as unittest

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

采用这种方式，如果需要转换到新版本的Python，并且不再需要unittest2模块时，可以简单的在测试模块中修改引入部分，而不需要修改其他的代码。

mock

`unittest.mock` 是Python中用于测试的一个库。在Python 3.3中已经成为了 标准库 的一部分。

对于旧版本的Python：

```
$ pip install mock
```

该库可以用模拟对象来替换待测试系统的一部分，你会从中知晓它们是如何被使用的。

例如，你可以给一个方法打猴子补丁：

```
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')
```

使用 `patch` 装饰器在待测试的模块中模拟类或者对象。在下述示例中，采用总是返回相同结果（仅限于该测试期间）的模拟对象来替代外部搜索系统。

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

# 这里的SearchForm指的是myapp中导入的类引用，并不是SearchFrom本身被引入前的所在
@mock.patch('myapp.SearchForm.search', mock_search)
def test_new_watchlist_activities(self):
    # get_search_results进行搜索操作并对结果进行迭代
    self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

Mock还有许多其他的配置方式来控制其行为。

日志

自Python 2.3之后，`logging` 模块已经成为标准库的一部分。**PEP 282** 里进对此行了简单的介绍。但该文档是出了名的难读，唯有 Python基础日志教程 相对容易学习。

日志主要有两个目的：

- **诊断性日志** 记录应用操作的相关事件。如果用户引入日志模块用来报告错误，那么就可以通过在日志中搜索上下文来获取相关错误信息。
- **审计性日志** 记录用于商业分析的事件。用户进行的事务可以被提取出来，然后与另外的用户详细信息组合形成报告，或者用来进行优化，以达到更好的业务目标。

日志还是Print输出?

只有在一种情况下 `print` 相比于日志是一个更好的选择：在命令行应用下展示帮助信息时。其他情况下，日志毫无疑问是更好的选择，原因如下：

- 每一个日志事件创建的 日志记录 包含有可读的诊断信息，例如文件名、完整路径、函数以及日志事件发生的行号。
- 除非你过滤掉所引用模块中的日志事件，否则它会被根logger自动访问，进而输出到应用程序本身的日志流中。
- 日志可以通过使用 `logging.Logger.setLevel()` 或者通过设置属性 `logging.Logger.disabled` 为 `True` 选择性不显示。

函数库中的日志

库开发中的日志配置 是 Python日志教程 中的一部分。由于是 **用户** 而不是库本身来说明当日志事件出现时究竟发生了什么，所以脑子里要不断重复下警告：

注解： 强烈建议除NullHandler外不要添加任何其他handler到所写库的日志logger中。

当在一个库中初始化loggers时，最佳实践方法是仅使用全局变量 `__name__` 来创建这些loggers：`logging` 模块使用点号来创建层次性的loggers，所以，使用 `__name__` 可以确保不会有名字冲突。

如下是来自 requests源码 的最佳实践示例 – 可以把这段代码放到你的 `__init__.py` 文件中：

```
# 设置默认的日志处理模块，避免产生 "No handler found" 警告。
import logging
try:  # Python 2.7+
    from logging import NullHandler
except ImportError:
    class NullHandler(logging.Handler):
        def emit(self, record):
            pass

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

应用程序中的日志

The Twelve-Factor App 是一份应用开发最佳实践的权威性参考，其中包含了一节 日志最佳实践 。它重点提倡把日志事件当作事件流来对待，然后把事件流发送到标准输出，以便被应用环境进行处理。

至少有三种方式来配置logger:

- **使用INI格式的文件**:
  - **优点**: 通过使用函数 `logging.config.listen()` 监听socket可以在程序运行时更新配置。
  - **缺点**: 相比在代码中配置logger，可控性较弱（比如 利用子类定制filters或者loggers）
- **使用字典或者JSON格式的文件**:
  - **优点**: 除了能在程序运行时更新外，还可以通过 `json` 模块载入文件来更新，该模块自Python 2.6进入标准库。
  - **缺点**: 相比在代码中配置logger，可控性较弱。
- **使用代码**:
  - **优点**: 对配置可进行完全控制。
  - **缺点**: 修改配置需要改动源代码。

使用INI文件配置的例子

这里我们把配置文件叫做 `logging_config.ini` 。该文件格式的更多细节可参考 Python日志教程 中的 配置日志 一节。

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
```

```
args=(sys.stderr,)

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

然后，在代码中调用 logging.config.fileConfig()：

```
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')
```

使用字典配置的例子

在Python 2.7中，可以使用包含有详细配置的字典。**PEP 391** 中列出了在配置字典中哪些元素是必备的，以及哪些元素是可选的。

```
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')
```

直接在代码中配置的例子

```
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')
```

常见的问题

很大程度上来讲，Python作为一门简洁一致的语言，会尽量避免一些让人感到惊讶的特性。然而，还是有一些情况会让新手感到迷惑不解。

这些情形中的一些是故意为之，但却可能会令人感到惊讶。一些则已被证明是语言的缺陷。通常，对于一系列难以捉摸的行为，乍一看很奇怪，但是一旦你明白这种惊讶背后的底层原因，你就会觉得合乎情理。

可变的默认参数

对于Python初学者而言，最为常见、看起来令人惊讶的就是Python对于函数定义中可变默认参数的处理。

你所写的代码

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

你所期望发生的

```
my_list = append_to(12)
print my_list

my_other_list = append_to(42)
print my_other_list
```

每次调用函数时，如果第二个参数没有提供，那么应该创建一个新的列表，所以输出应该是:

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

事实上发生的

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

一旦函数被定义后，新的列表就会被创建，后续的每次函数调用都会使用同一个列表。

当函数被定义时，Python的默认参数就会被求值，而不是发生在每次函数调用时（这和Ruby是一样的）。这就意味着，如果你使用了可变的默认参数且改变该参数，你 将会 并且已经无意识的修改了之后所有调用该函数时的参数对象。

如何处理这种情况

每次函数调用时，通过使用一个默认的参数值（`None` 通常是一个好的选择）来提示该调用未提供参数，这时再创建一个新的对象。

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

当问题就不再是问题

有时候你可以专门“利用”（注：按所预期的使用）这个行为来维护函数调用之间的状态。通常在编写缓存函数的时候会用到这个特性。

延迟绑定闭包

另外一个迷惑源自Python在闭包中绑定变量的方式（或在周围全局作用域）。

你所写的代码

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

你所期望发生的

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

包含有5个函数的列表，每个函数拥有自己的封闭的 `i` 变量来与其参数进行乘运算，产生结果如下：

```
0
2
4
6
8
```

事实上发生的

```
8
8
8
8
8
```

5个函数都已创建，然而这些函数却都把 `x` 乘以4。

Python的闭包采用 延迟绑定 。这意味着，闭包中使用到的变量，它的值是在内部函数被调用时才会进行查找。

这里例子中，无论何时，当所返回函数中的 任何 一个进行调用时， `i` 的值只有在调用时刻才在周边作用域中进行查找。而此刻，循环操作早已结束且 `i` 最后的值已变为4。

对于这个问题，更糟糕的一点是对这个结果的广泛误解：人们以为这与Python中的 lambdas 有关。其实，使用 `lambda` 表达式创建的函数没有任何特殊，事实上，使用常用的 `def` 创建的函数同样存在这个问题。

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

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

    return multipliers
```

如何处理这种情况

最为常用的解决方案可能需要一点hack。多亏前面提及的关于函数参数默认值求值问题（参见 可变的默认参数），你可以创建一个闭包，然后利用默认的参数值立刻绑定其参数，就像下面这样：

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

另外一个方案，你可以使用函数 functools.partial：

```
from functools import partial
from operator import mul

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

当问题就不再是问题

有时候，你是希望闭包可以按照这种延迟绑定的行为来执行的。在很多情形下，延迟绑定是非常有用的。当然，很不幸，通过循环来创建唯一函数成了一个反例的情形。

无处不在的字节码(.pyc)文件

默认情况下，当从一个文件执行Python代码的时候，Python解释器会自动在磁盘上产生该文件的字节码文件，例如： `module.pyc` 。

这些 `.pyc` 文件不应该提交到源码的版本库中。

从理论上讲，由于性能的原因，这种行为默认是开启的。因为如果没有这些字节码文件，每次源码文件被载入执行时，都需要重新生成字节码。

禁用字节码(.pyc)文件

幸运的是，产生字节码的过程极其快速，所以在开发代码的时候并不需要担心这些。

当然，这些字节码文件相当的烦人，所以可以通过下面的方法来摆脱它们：

```
$ export PYTHONDONTWRITEBYTECODE=1
```

一旦设置 `$PYTHONDONTWRITEBYTECODE` 环境变量，Python就不会再产生字节码文件，这样你的开发环境可以保持干净整洁。

我建议在你的 `~/.profile` 文件中设置这个环境变量。

移除字节码(.pyc)文件

如果已经存在字节码文件，下述命令可以移除这些文件:

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

在项目的跟目录下执行这行命令，所有的 `.pyc` 文件瞬间就会消失的无影无踪。6不6?

选择许可证

源码发布 需要 一个许可证。在灯塔国，如果没有指明许可证，用户是没有合法权利进行下载、修改或者再分发的。进一步讲，人们也无法向你的代码进行贡献，除非你告诉他们应该遵守什么规则。挑选合适的许可证比较复杂，这里给出一些指点:

开源。有大量的 开源许可证 可供选择。

通常，这些许可证属于以下这两类中的一种:

1. 许可证更多地聚焦于用户按照其意愿使用软件的自由（这类许可证更加宽容，例如MIT、BSD以及Apache）。
2. 另一类许可证更多地聚焦于确保代码本身 — 包括对代码任何的修改和分发 — 总是保持自由开放（这类不那么宽容的许可证包括GPL以及LGPL）。

后一种许可证不允许用户在软件上添加代码却在分发时不包含这部分更改的源码（译者注：即作出的更改必须也开源），就这点而言，显得不太宽容。

为了帮助你给自己的项目选择一个合适的许可证，这里有一份 如何选择许可证，**参考下吧**。

更加宽容的许可证

- PSFL (Python Software Foundation License) – 用于Python本身
- MIT / BSD / ISC
  - MIT (X11)
  - New BSD
  - ISC
- Apache

不那么宽容的许可证

- LGPL
- GPL
  - GPLv2
  - GPLv3

tl;drLegal 是一份许可证概览，里面解释了用户在使用一个特定软件时能做什么、不能做什么以及必须做什么。

CHAPTER 3

Python应用场景

本章主要聚焦于不同场景下工具和模块的使用建议。

网络应用

HTTP

超文本传输协议是一种用于分布式、协作化、超媒体信息系统的应用协议。HTTP是万维网中数据交互的基础。

Requests

Python标准库中的urllib2模块提供了你可能会用到的绝大部分HTTP功能，然而，该模块的API却十分难用。该模块是为不同时间段、不同web构建的。所以，即使要完成一个很简单的任务也需要大量的工作（甚至需要重写一些方法）。

Requests做了Python HTTP的所有工作 — 可以无缝的集成web服务。通过使用Requests，不再需要人工添加查询字符串到URL中，又或者编码POST数据。Keep-alive以及HTTP连接池都是100%的自动完成，这是通过内嵌在Requests中的urllib3来完成的，

- 参考文档
- PyPi
- GitHub

分布式系统

ZeroMQ

ØMQ（也写作ZeroMQ、0MQ或ZMQ）是一个高性能的异步消息库，旨在用于高扩展性的分布式或并发应用中。它提供了一个消息队列，但与面向消息的中间件有所不同， ØMQ系统不需要特定的消息协商器就可以运行。库本身的设计与socket API有着相似的接口。

RabbitMQ

RabbitMQ是一个实现了高级消息队列协议（AMQP）的开源消息协商器软件。RabbitMQ服务器采用Erlang编程语言实现，并且构建在开放电信平台（OTP）上来达到集群化和容错。所有主流的编程语言都实现了与协商器交互的客户端。

- 主页
- GitHub组织

Web应用及框架

作为一种适用于快速原型以及大型项目的强大脚本语言，Python在Web应用开发中被广泛使用。

上下文环境

WSGI

Web服务器网关接口（缩写为"WSGI"）是Python web应用框架与web服务器交互的标准接口。通过标准化Python web框架与web服务器之间的行为和交互，WSGI使得编写可部署于任何 *WSGI*兼容服务器 上的可移植性Python代码变成了可能。WSGI的相关文档见 **PEP 3333** 。

框架

大体上来说，web框架包含了各种库的集合，以及用于定制代码实现web应用（比如一个交互式的web站点）的主处理器。大部分的web框架包含了各种模式以及工具来实现以下几点：

URL路由 匹配到来的HTTP请求到特定的Python代码来执行相应逻辑

请求响应对象 对接收自或发送给用户浏览器的信息进行包装

模板引擎 用于分离实现应用逻辑的Python代码和产生的HTML（或其他形式）输出

开发时使用的web服务器 在开发机上运行HTTP服务器可以进行快速开发；当文件更新时，通常会自动重载服务端代码。

Django

Django 是一个"内置电池"的web应用框架，对于创建面向内容的站点来说是一个极好的选择。通过提供大量开箱即用的工具和模式，Django旨在鼓励以最佳实践方式编写代码的同时，可以快速构建复杂的、数据库驱动的web应用。

Django有着一个活跃的大型社区，有许多预先构建好的 可复用模块 可以用来构建新的项目，抑或通过定制来达到你的要求。

在 美国 和 欧洲 每年都有Django的会议。

当今大多数新的Python web应用都是使用Django来构建的。

Flask

Flask 是Python中的一个 “微框架”，对于构建小型的应用、API或者web服务，这是一个极好的选择。

使用Flask构建应用在很大程度上就和编写标准Python模块差不多，除了要绑定路由到一些函数之上。这种方式棒极了。

Flask并没有向你提供可能会用到的所有功能，而是实现了web应用框架中大部分常用的核心组件，比如URL路由、请求响应对象和模板。

如果采用Flask，你可以根据自己爱好来为你的应用选择其他任何组件。比如，Flask并没有内建数据库访问或者表单的生成与验证。

这样做其实很好，因为很多web应用并不需要这些特性。对于那些需要这些特性的项目而言，有许多 扩展 可以满足你的需求。又或者你可以使用任何你想用的库！

对于那些不太适用Django的Python web应用来说，Flask是默认的选择。

Tornado

Tornado 是Python中的一个异步web框架，有着自己的事件循环，这使得其本身可以支持WebSockets。优秀的tornado应用以性能卓越而著称。

我不太建议使用Tornado，除非你认为非用不可。

Pyramid

Pyramid 是一个聚焦于模块化、极具伸缩性的框架。框架本身内建有少量的库（“电池”），并鼓励用户扩展这些基础功能。

不像Django和Flask，Pyramid用户基础并不大。本身是一个功能很强大的框架，但是对于如今新的Python web应用来说并不是一个流行的选择。

Web服务器

Nginx

Nginx （发音 “engine-x”）是一个web服务器以及HTTP、SMTP和其他协议的反向代理。以高性能、相对简单及和许多应用服务器的（如WSGI服务器）兼容性著称。它包含有大把的特性，比如负载均衡、基本认证、流处理以及其他特性。Nginx设计用以服务于高负载的站点，正在逐渐变得越来越流行。

WSGI服务器

独立的WSGI服务器往往比传统的web服务器使用更少的资源，并且提供更高的性能 [3] 。

Gunicorn

Gunicorn （Green Unicorn）是一个用来驱动Python应用的WSGI服务器，纯Python实现。与其他Python web服务器不同，它有着非常体贴的接口，并且非常易用和配置。

Gunicorn有着很明智合理的默认配置。然而，一些其他服务器，例如uWSGI，虽然有着惊人的可定制性，但也因此变得更加难以高效使用。

[3] Python WSGI服务器基准测试

对于如今新的Python web应用而言，Gunicorn是更加推荐的选择。

Waitress

Waitress 也是一个纯Python的WSGI服务器，声称有着“非常可接受的性能”。其文档并不太详细，但是确实提供了一些Gunicorn没有提供的优秀功能（如HTTP请求缓冲）。

Waitress在Python web开发社区正在变得越来越流行。

uWSGI

uWSGI 为构建托管服务提供了全栈的功能。除了进程管理、进程监控以及其他功能，uWSGI还可以作为多种编程语言和协议的应用服务器 - 当然包括Python和WSGI。uWSGI既可以作为独立的web路由器来运行，也可以运行在全功能的web服务器（比如Nginx和Apache）后端。后一种情况下，web服务器可以配置uWSGI与应用之间采用 uwsgi协议 来通信。uWSGI的web服务器支持通过传递环境变量和进一步的调整来动态配置Python。更详细的信息，参见 uWSGI魔法变量 。

我不建议使用uWSGI，除非你知道为什么需要使用。

服务端最佳实践

当前主流部署Python应用的方式主要是采用如 *Gunicorn* 的WSGI服务器，然后直接或间接的放置在轻量级web服务器，如 *nginx* 后面。

WSGI服务器主要是用于Python应用的处理，与此同时，web服务器则处理更适合它的任务，比如静态文件服务、请求路由、DDoS保护以及基本认证。

托管部署

平台即服务（PaaS）是一种云计算基础设施的类型，会抽象并管理基础设施、路由以及web应用的扩展。使用PaaS时，应用开发者可以聚焦于编写应用代码而不用关心部署的细节。

Heroku

Heroku 对Python 2.7-3.5 应用提供了第一流的支持。

Heroku支持所有类型的Python web应用、服务器以及框架。可以在Heroku上免费的开发应用。一旦应用可以用于生产，你可以升级到兴趣或专业应用。

Heroku维护了 详细的文章 来说明如何在Heroku上使用Python，同时也有一份 手把手指南 来帮助设置你的第一个应用。

Heroku是当前比较推荐的用于部署Python web应用的PaaS。

Eldarion

Eldarion （之前叫做Gondor）由Kubernetes、CoreOS和Docker构建的PaaS平台。该平台支持任何WSGI应用，并且有一份指南说明如何部署 Django项目 。

模板

大多数WSGI应用会以HTML或其他标记语言作为HTTP请求的响应。关注点分离的概念建议我们采用模板，而不是直接从Python中产生文本内容。模板引擎管理着一系列的模板文件，采用层次系统和包含系统来避免不必要的重复，并负责渲染（生成）实际的内容，利用应用产生的动态内容来填充模板的静态内容。

由于模板文件有时会由设计人员或者前端开发者来编写，因此处理起不断增长的复杂性会很困难。

对于如何把应用中的动态内容传递给模板引擎和模板本身，有一些通用且好的实践可供参考：

- 应当只把渲染时必要的动态内容传递给模板文件。避免传递额外的内容"以防万一"：添加缺失的变量远比移除一个不用的变量要容易。
- 许多模板引擎允许在模板中使用复杂的语句或赋值，并且还允许在模板中调用Python代码。这种便捷会导致复杂性的不可控，进而使得很难找到bug。
- 通常情况下需要把Javascript模板与HTML模板混合。对于这种设计来说，一个明智的方式是把需要由HTML模板传递变量内容到JavaScript代码的部分隔离。。

Jinja2

Jinja2 是一个得到普遍好评的模板引擎。

它采用基于文本的模板语言，这样就可以用于生成任何类型的标记语言，不仅仅是HTML。Jinja2允许定制过滤器、标签、测试以及全局内容。相比于Django的模板系统，Jinja2有许多的改进。

这里是Jinja2中一些重要的HTML标签:

```
{# 这是注释 #}

{# 下一个标签是变量输出 #}
{{title}}

{# 块标签，通过继承可以被其他html代码替换 #}
{% block head %}
<h1>This is the head!</h1>
{% endblock %}

{# 迭代输出数组 #}
{% for item in list %}
<li>{{ item }}</li>
{% endfor %}
```

接下来的代码清单是一个与Tornado服务器结合的web站点示例。Tornado使用起来并不复杂。

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

# 导入Tornado
import tornado.ioloop
import tornado.web

# 载入模板文件 templates/site.html
TEMPLATE_FILE = "site.html"
templateLoader = FileSystemLoader( searchpath="templates/" )
templateEnv = Environment( loader=templateLoader )
template = templateEnv.get_template(TEMPLATE_FILE)

# 用于渲染的著名电影列表
```

```
movie_list = [[1,"The Hitchhiker's Guide to the Galaxy"],[2,"Back to future"],[3,"Matrix"]]

# template.render() 返回包含有渲染后html的字符串
html_output = template.render(list=movie_list,
                              title="Here is my favorite movie list")

# 主页的处理器
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        # 返回渲染后的模板字符串到浏览器
        self.write(html_output)

# 设定路由   (127.0.0.1:PORT/)
application = tornado.web.Application([
    (r"/", MainHandler),
])
PORT=8884
if __name__ == "__main__":
    # 设置服务器
    application.listen(PORT)
    tornado.ioloop.IOLoop.instance().start()
```

base.html 文件可以作为所有站点页面的基础，这些站点页面实现其中的内容块即可。

```
<!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">
    {# 下一行会由site.html模板中的内容填充 #}
    {% block content %}{% endblock %}
</div>
<div id="footer">
    {% block footer %}
    &copy; Copyright 2013 by <a href="http://domain.invalid/">you</a>.
    {% endblock %}
</div>
</body>
```

下一代码清单是我们Python应用载入的的站点页面（site.html），该页面扩展了 base.html 。内容块会自动嵌入到 base.html 页面对应的块中。

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

```
{% endblock %}
```

对于新的Python web应用，Jinja2是比较推荐的模板库。

Chameleon

Chameleon 页面模板是一个HTML/XML模板引擎，该引擎实现了 模板属性语言 (TAL) 、 TAL表达式语法 (TALES) 和 宏扩展TAL (Metal) 语法。

Chameleon可用于Python 2.5及以上版本（包括3.x和pypy），常用于 Pyramid框架 。

页面模板会在你的文档结构中添加特殊的元素属性和文本标记。通过使用一组简单的语言构造，你可以控制文档流、元素的重复、文本的替换和转化。由于是基于属性的语法，未渲染的页面模板是合法的HTML，所以可以在浏览器中查看，也可以在WYSIWYG的编辑器中编辑。这使得与设计师来回的协作，以及在浏览器中使用静态文件构建原型变得更加容易。

基本的TAL语言可以很容易的从下面的例子中了解：

```
<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>
```

** 是插入文本的模式，它是如此常见，以至于当你不需要保证未渲染的模板严格合法时，你可以使用更加简短可读的语法来替换，即 *${expression}* ，具体如下：

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

但是请记住完整的 *Default Text* 语法还允许在未渲染的模板中包含默认内容。

由于Chameleon来自Pyramid世界，所以并未广泛使用。

Mako

Mako 是一种会编译为Python的模板语言，以达到性能最大化。它的语法和API借鉴于其他模板语言（比如Django和Jinja2）最好的部分，它是 Pylons and Pyramid web框架包含的默认模板语言。

Mako模板示例如下：

```
<%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>
```

要想渲染一个最基本的模板，你可以这样做：

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

在Python web社区，Mako备受推崇。

参考

HTML页面爬取

网页爬取

网站采用HTML编写，这意味着每个网页都是一个结构化的文档。有时候我们往往需要从这些网页中获取一些数据，与此同时还要保留其结构。网站一般不会以合适的格式来提供他们的数据，比如采用 `csv` 或者 `json` 。

这时，网页爬取就派上了用场。网页爬取是一种采用电脑程序在网页上筛选并收集数据的行为，这些数据以对你最为有用的格式来保存，同时还保留了其结构。

lxml与Requests

lxml 是一个用来快速解析XML和HTML文档且具有良好扩展性的库，甚至可以处理各种杂乱的标签。我们还会使用 Requests 模块来替代内置的urllib2模块，以便提高速度和可读性。你可以很容易的使用命令 `pip install lxml` 和 `pip install requests` 来安装。

首先，导入这两个库：

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

接下来我们将使用 `requests.get` 来获取包含我们所需数据的网页，采用 `html` 模块来解析，并把结果保存在 `tree` 中。

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

我们应该使用 `page.content` 而不是 `page.text`，因为 `html.fromstring` 默认使用 `bytes` 来作为输入。

`tree` 现在以良好的树状结构包含整个HTML文件，我们可以采用两种方式来遍历：XPath和CSSSelect。在这个示例中，我们主要介绍前一种。

XPath是一种在HTML或XML等结构化文档中定位信息的方式。 W3Schools 上有对XPath很好的介绍。

此外还有各种各样的工具来获取元素的XPath，比如Firefox中的FireBug或者Chrome中的Inspector。如果你正在使用Chrome，你可以右击元素，选择“检查”，高亮对应的代码，再次右击然后选择 “Copy -> Copy XPath”。

快速分析下面代码后，可以发现在我们的页面中，数据包含在两个元素之中 - 一个是title为”buyer-name”的div块，另外一个是class为”item-price”的span。

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

知道这些后，我们可以构造一个正确的XPath查询，并且按下列方式使用lxml中的 `xpath` 函数：

```
# 此处会构造一个购买者的列表：
buyers = tree.xpath('//div[@title="buyer-name"]/text()')
# 此处会构造一个价格的列表：
prices = tree.xpath('//span[@class="item-price"]/text()')
```

我们来看看得到的确切数据是什么：

```
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']
```

恭喜恭喜！我们已经成功的使用lxml和Requests从网页中爬取到了所有我们想要的数据。目前这些数据以两个列表的形式保存在内存中。这时候就可以在这些数据上做各种很酷的操作了：我们可以使用Python来分析这些数据或者把这些数据保存在一个文件中，然后分享给其他人。

可以想一些更加酷的idea，比如修改这个脚本来迭代处理网页剩下的部分，或者使用线程重写该应用来提高速度。

命令行应用

命令行应用也叫 控制台应用 ，是为文本界面设计的电脑程序， 比如 shell 。命令行应用通常接收多个输入作为参数或者子命令， 同时还会接收一些选项来作为标识或开关。

一些流行的命令行应用包括：

- Grep - 文本数据搜索工具
- curl - 采用URL语法进行数据传输的工具
- httpie - HTTP客户端命令行工具，是cURL更为友好的替代品

- git - 一种分布式版本控制系统
- mercurial - 一种分布式版本控制系统，主要采用Python开发

Clint

clint 是一个用于开发命令行应用的Python模块，包含很多有用的工具， 支持的特性包括：命令行颜色/缩排、简单强大的列输出、基于迭代器的进度条以及隐式参数处理等。

Click

click (Command-line Interface Creation Kit，首字母缩写) 是一个使用尽可能少的代码来实现命令行接口的Python包，该包采用了组合方式来实现。这个“命令行接口构造工具”具有开箱即用的默认设置，同时不失高可配置性。

docopt

docopt 是一个轻量且Pythonic的包。通过解析POSIX风格的指令来创建命令行接口，很直观易用。

Plac

Plac 是对Python标准库中的 argparse 的简单包装， 采用声明式接口（即通过推断而不是手写命令的方式来构建出参数解析器）来隐藏其复杂性。其目标用户包括：初级用户、 程序员、系统管理员、科学家以及那些编写用完即扔脚本的人，该工具可以帮助他们快速简单的创建命令行接口。

Cliff

Cliff 是一个构建命令行程序的框架。它采用了setuptools的入口点 来提供子命令、格式化输出以及其他扩展。该框架目的是用于创建包含多层级的命令，比如subversion和git，这些程序会处理某个 基本参数的解析，然后再调用对应的子命令来进行具体的工作。

Cement

Cement 是一个先进的命令行应用框架，其目标是在不牺牲高品质的前提下，引入一个标准化、特性齐全的平台， 该平台对于各种复杂程度的命令行应用提供良好支持，同时还能满足快速开发。Cement具有高伸缩性，使用场景就可以满足微框架的简单性，也可以 满足大型框架的复杂性。

GUI应用

按字母排序的GUI应用列表。

Camelot

Camelot 受Django管理界面的启发，在Python、SQLAlchemy和Qt的基础上提供了各种组件来构建应用。

可用的参考资源主要是其网站：http://www.python-camelot.com 和邮件列表: https://groups.google.com/forum/#!forum/project-camelot

Cocoa

注解： Cocoa框架仅用于OS X，如果要编写跨平台的应用就不要考虑了。

GTK

PyGTK提供了对GTK+工具集的Python绑定。与GTK+库本身一样，也采用了GNU LGPL许可证。需要注意的是PyGTK目前只支持GTK-2.X的API（不支持GTK-3.0）。对于新项目来说目前已不推荐使用PyGTK，现有的PyGTK应用也推荐迁移到PyGObject上。

PyGObject 又叫(PyGi)

PyGObject 提供了整个GNOME软件平台的Python绑定，且与GTK+ 3完全兼容。这里有一份入门资料 Python GTK+ 3指南 。

API参考

Kivy

Kivy 是一个Python库，可用于开发多点触屏的富媒体应用。其目标是为了能够进行快速轻松的交互设计及快速原型，同时保证代码的可重用性和可部署性。

Kivy采用Python编写，基于OpenGL，支持多种输入设备，例如：鼠标、双向鼠标、TUIO触摸协议、Wii控制器、Windows的WM_TOUCH消息、HID触摸以及苹果公司的产品等等。

Kivy由一个社区进行开发，非常活跃且免费使用，可在所有主流平台（Linux, OSX, Windows, Android）上使用。

主要资源可以在其网站上找到：http://kivy.org

PyObjC

注解： Cocoa框架仅用于OS X，如果要编写跨平台的应用就不要考虑了。

PySide

PySide是对跨平台GUI工具Qt的Python绑定。

```
pip install pyside
```

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

PyQt

注解： 如果你的软件没有完全遵从GPL，那么你需要购买商业许可证！

PyQt提供了Qt框架的Python绑定（见后面）。

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

PyjamasDesktop (pyjs Desktop)

PyjamasDesktop是Pyjamas的移植。PyjamasDesktop是一组用于桌面及跨平台框架的组件集，v0.6版本之后，PyjamasDesktop成了Pyjamas(Pyjs)的一部分。 简单来说，就是可以采用与Python Web应用完全相同的代码但是作为独立桌面应用来执行。

PyjamasDesktop的Python Wiki 。

主页： pyjs Desktop 。

Qt

Qt 是一个广泛使用的跨平台应用框架，可用于开发GUI以及非GUI应用。

Tk

Tkinter是Tcl/Tk之上很薄的面向对象包装层。 **可以使用Python标准库的优势使得它成为最方便且兼容性良好编程工具集** 。

Tk和Tkinter二者都可以在大多数的Unix平台使用，当然Windows及Macintosh系统也同样支持。从8.0版本开始，Tk在所有平台提供了原生界面的支持。

TkDocs 上有一份非常不错的多语言Tk教程，包含了Python的示例。更多信息见 Python Wiki 。

wxPython

wxPython是一个Python语言的GUI工具集。可以让Python程序员很简便的创建出健壮、功能丰富的图形用户界面。 它是一个Python的扩展模块（原生代码），通过包装著名的跨平台C++ GUI库wxWidgets来实现。

安装wxPython： 到 *http://www.wxpython.org/download.php#stable* 下载适合你所使用操作系统的包。

数据库

DB-API

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

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

SQLAlchemy

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

```
$ pip install sqlalchemy
```

Records

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

```
$ pip install records
```

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

Django ORM

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

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

The basics:

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

peewee

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

PonyORM

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

SQLObject

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

网络

Twisted

Twisted 是一个事件驱动的网络引擎。可以用来构建基于多种不同网络协议的应用程序，包括http服务器/客户端、使用SMTP/POP3/IMAP或SSH协议的应用、即时通信应用以及 更多 。

PyZMQ

PyZMQ 是 ZeroMQ 的Python绑定。ZeroMQ是一个高性能的异步消息库，其中一个最大的优点是采用了无代理的方式来处理消息队列。ZeroMQ的基本模式如下：

- 请求-应答: 把一组客户端关联到一组服务上。这属于一种远程过程调用及任务分发模式。
- 发布-订阅: 把一组发布者关联到一组订阅者上。这是一种数据分发模式。
- 推送-拉取（或管道）：采用多阶段及循环输出/输入模式把一组节点关联起来。这是一种并行任务分发及收集模式。

快速上手参见 ZeroMQ指南 。

gevent

gevent 是一个基于协程的Python网络库，在libev事件循环上利用greenlets封装了更加高层的同步API。

Systems Administration

Fabric

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

Install Fabric:

```
$ pip install fabric
```

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

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

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

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

@task
def deploy():
    with cd('/var/www/project-env/project'):
        with prefix('. ../bin/activate'):
```

```
            run('git pull')
            run('touch app.wsgi')
```

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

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

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

and we can deploy with:

```
$ fab deploy
```

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

> Fabric Documentation

Salt

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

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

```
$ pip install salt
```

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

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

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

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

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

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

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

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

```
apache:
  pkg:
    - installed
```

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

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

> Salt Documentation

Psutil

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

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

```
# Functions to get system values:
from psutil import cpu_percent, net_io_counters
# Functions to take a break:
from time import sleep
# Package for email services:
import smtplib
import string
MAX_NET_USAGE = 400000
MAX_ATTACKS = 4
attack = 0
counter = 0
while attack <= MAX_ATTACKS:
    sleep(4)
    counter = counter + 1
    # Check the cpu usage
    if cpu_percent(interval = 1) > 70:
        attack = attack + 1
    # Check the net usage
    neti1 = net_io_counters()[1]
    neto1 = net_io_counters()[0]
    sleep(1)
    neti2 = net_io_counters()[1]
    neto2 = net_io_counters()[0]
    # Calculate the bytes per second
    net = ((neti2+neto2) - (neti1+neto1))/2
    if net > MAX_NET_USAGE:
        attack = attack + 1
    if counter > 25:
        attack = 0
        counter = 0
# Write a very important email if attack is higher than 4
TO = "you@your_email.com"
FROM = "webmaster@your_domain.com"
SUBJECT = "Your domain is out of system resources!"
text = "Go and fix your server!"
BODY = string.join(("From: %s" %FROM,"To: %s" %TO,"Subject: %s" %SUBJECT, "",text), "\r\n")
server = smtplib.SMTP('127.0.0.1')
server.sendmail(FROM, [TO], BODY)
server.quit()
```

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

Ansible

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

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

```
$ pip install ansible
```

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

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

```
[server_name]
127.0.0.1
```

Here is an example playbook: `ping.yml`

```
---
- hosts: all

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

To run the playbook:

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

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

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

Chef

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

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

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

To create a simple cookbook the knife command is used:

```
knife cookbook create cookbook_name
```

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

- Chef Documentation

Puppet

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

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

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

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

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

```
$ facter kernel
Linux
```

```
$ facter operatingsystem
Ubuntu
```

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

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

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

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

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

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

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

file { '/etc/ssh/sshd_config':
    source   => 'puppet:///modules/sshd/sshd_config',
    owner    => 'root',
    group    => 'root',
    mode     => '640',
```

```
    notify   =>  Service['sshd'], # sshd will restart
                                  # whenever you edit this
                                  # file
    require  => Package['openssh-server'],

}

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

For more information, refer to the Puppet Labs Documentation

Blueprint

待处理

Write about Blueprint

Buildout

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

Shinken

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

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

Continuous Integration

Why?

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

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

Jenkins

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

Buildbot

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

Tox

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

- Checking that packages install correctly with different Python versions and interpreters
- Running tests in each of the environments, configuring your test tool of choice
- Acting as a front-end to Continuous Integration servers, reducing boilerplate and merging CI and shell-based testing.

Travis-CI

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

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

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

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

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

Speed

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

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

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

```
# CPython
$ ./python -V
Python 2.7.1
$ ./python measure2.py
1.06774401665
1.45412397385
1.51485204697
1.54693889618
1.60109114647
```

Context

The GIL

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

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

C Extensions

The GIL

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

C Extensions

Cython

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

Here's an example of strong typing with Cython:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

These lines both need a remark:

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

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

```
Cython time: 0.0054 seconds

Python time: 0.0566 seconds
```

And here the output of an embedded ARM beaglebone machine:

```
Cython time: 0.0196 seconds

Python time: 0.3302 seconds
```

Pyrex

Shedskin?

Concurrency

Concurrent.futures

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

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

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

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

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

```
from concurrent.futures import ThreadPoolExecutor
import requests

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

pool = ThreadPoolExecutor(max_workers=5)

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

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

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

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

cancel() Attempt to cancel the call.

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

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

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

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

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

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

```
from concurrent.futures import ProcessPoolExecutor, as_completed

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

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

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

futures = []
with ProcessPoolExecutor(max_workers=4) as pool:
    # Schedule the ProcessPoolExecutor to check if a number is prime
    # and add the returned Future to our list of futures
    for p in PRIMES:
        fut = pool.submit(is_prime, p)
        futures.append(fut)

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

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

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

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

Threading

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

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

```
from threading import Thread
import requests

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

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

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

```
some_thread.join()
```

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

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

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

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

```
from threading import Lock, Thread

file_lock = Lock()

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

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

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

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

Spawning Processes

Multiprocessing

Scientific Applications

Context

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

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

Tools

IPython

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

Libraries

NumPy

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

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

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

Numba

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

SciPy

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

Matplotlib

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

Pandas

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

Rpy2

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

PsychoPy

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

Resources

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

Unofficial Windows Binaries for Python Extension Packages

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

Anaconda

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

Canopy

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

Image Manipulation

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

A brief description of both is given below.

Python Imaging Library

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

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

Installation

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

After that, it's straightforward:

```
$ pip install Pillow
```

Example

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

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

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

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

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

OpenSource Computer Vision

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

Installation

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

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

```
$ pip install numpy
```

Example

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

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

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

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

Data Serialization

What is data serialization?

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

Pickle

The native data serialization module for Python is called Pickle.

Here's an example:

```
import pickle

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

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

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

Protobuf

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

XML parsing

untangle

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

For example, an XML file like this:

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

can be loaded like this:

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

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

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

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

xmltodict

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

An XML file like this:

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

can be loaded into a Python dict like this:

```
import xmltodict

with open('path/to/file.xml') as fd:
    doc = xmltodict.parse(fd.read())
```

and then you can access elements, attributes and values like this:

```
doc['mydocument']['@has'] # == u'an attribute'
doc['mydocument']['and']['many'] # == [u'elements', u'more elements']
doc['mydocument']['plus']['@a'] # == u'complex'
doc['mydocument']['plus']['#text'] # == u'element as well'
```

xmltodict also lets you roundtrip back to XML with the unparse function, has a streaming mode suitable for handling files that don't fit in memory and supports namespaces.

JSON

The json library can parse JSON from strings or files. The library parses JSON into a Python dictionary or list. It can also convert Python dictionaries or lists into JSON strings.

Parsing JSON

Take the following string containing JSON data:

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

It can be parsed like this:

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

and can now be used as a normal dictionary:

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

You can also convert the following to JSON:

```
d = {
    'first_name': 'Guido',
    'second_name': 'Rossum',
    'titles': ['BDFL', 'Developer'],
}

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

simplejson

The JSON library was added to Python in version 2.6. If you're using an earlier version of Python, the simplejson library is available via PyPI.

simplejson mimics the json standard library. It is available so that developers that use older versions of Python can use the latest features available in the json lib.

You can start using simplejson when the json library is not available by importing simplejson under a different name:

```
import simplejson as json
```

After importing simplejson as json, the above examples will all work as if you were using the standard json library.

Cryptography

Cryptography

Cryptography

Cryptography is an actively developed library that provides cryptographic recipes and primitives. It supports Python 2.6-2.7, Python 3.3+ and PyPy.

Cryptography is divided into two layers of recipes and hazardous materials (hazmat). The recipes layer provides simple API for proper symmetric encryption and the hazmat layer provides low-level cryptographic primitives.

Installation

```
$ pip install cryptography
```

Example

Example code using high level symmetric encryption recipe:

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"A really secret message. Not for prying eyes.")
plain_text = cipher_suite.decrypt(cipher_text)
```

PyCrypto

PyCrypto is another library, which provides secure hash functions and various encryption algorithms. It supports Python version 2.1 through 3.3.

Installation

```
$ pip install pycrypto
```

Example

```
from Crypto.Cipher import AES
# Encryption
encryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
cipher_text = encryption_suite.encrypt("A really secret message. Not for prying eyes.")

# Decryption
decryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
plain_text = decryption_suite.decrypt(cipher_text)
```

Machine Learning

Python has a vast number of libraries for data analysis, statistics and Machine Learning itself, making it a language of choice for many data scientists.

Some widely used packages for Machine Learning and other Data Science applications are enlisted below.

Scipy Stack

The Scipy stack consists of a bunch of core helper packages used in data science, for statistical analysis and visualising data. Because of its huge number of functionalities and ease of use, the Stack is considered a must-have for most data science applications.

The Stack consists of the following packages (link to documentation given):

1. NumPy
2. SciPy library
3. Matplotlib
4. IPython
5. pandas
6. Sympy
7. nose

The stack also comes with Python bundled in, but has been excluded from the above list.

Installation

For installing the full stack, or individual packages, you can refer to the instructions given here.

NB: Anaconda is highly preferred and recommended for installing and maintaining data science packages seamlessly.

scikit-learn

Scikit is a free and open-source machine learning library for Python. It offers off-the-shelf functions to implement many algorithms like linear regression, classifiers, SVMs, k-means, Neural Networks etc. It also has a few sample datasets which can be directly used for training and testing.

Because of its speed, robustness and easiness to use, it's one of the most widely-used libraries for many Machine Learning applications.

Installation

Through PyPI:

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

Through conda:

```
conda install scikit-learn
```

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

Example

For this example, we train a simple classifier on the Iris dataset, which comes bundled in with scikit-learn.

The dataset takes four features of flowers: sepal length, sepal width, petal length and petal width, and classifies them into three flower species (labels): setosa, versicolor or virginica. The labels have been represented as numbers in the dataset: 0 (setosa), 1 (versicolor) and 2 (virginica).

We shuffle the Iris dataset, and divide it into separate training and testing sets: keeping the last 10 data points for testing and rest for training. We then train the classifier on the training set, and predict on the testing set.

```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.metrics import accuracy_score
import numpy as np

#loading the iris dataset
iris = load_iris()

x = iris.data #array of the data
y = iris.target #array of labels (i.e answers) of each data entry

#getting label names i.e the three flower species
y_names = iris.target_names

#taking random indices to split the dataset into train and test
test_ids = np.random.permutation(len(x))

#splitting data and labels into train and test
#keeping last 10 entries for testing, rest for training

x_train = x[test_ids[:-10]]
x_test = x[test_ids[-10:]]

y_train = y[test_ids[:-10]]
y_test = y[test_ids[-10:]]

#classifying using decision tree
clf = tree.DecisionTreeClassifier()

#training (fitting) the classifier with the training set
clf.fit(x_train, y_train)

#predictions on the test dataset
pred = clf.predict(x_test)

print pred #predicted labels i.e flower species
print y_test #actual labels
print (accuracy_score(pred, y_test))*100 #prediction accuracy
```

Since we're splitting randomly and the classifier trains on every iteration, the accuracy may vary. Running the above code gives:

```
[0 1 1 1 0 2 0 2 2 2]
[0 1 1 1 0 2 0 2 2 2]
100.0
```

The first line contains the labels (i.e flower species) of the testing data as predicted by our classifier, and the second line contains the actual flower species as given in the dataset. We thus get an accuracy of 100% this time.

More on scikit-learn can be read in the documentation.

Interfacing with C/C++ Libraries

C Foreign Function Interface

CFFI provides a simple to use mechanism for interfacing with C from both CPython and PyPy. It supports two modes: an inline ABI compatibility mode (example provided below), which allows you to dynamically load and run functions from executable modules (essentially exposing the same functionality as LoadLibrary or dlopen), and an API mode, which allows you to build C extension modules.

ABI Interaction

```
from cffi import FFI
ffi = FFI()
ffi.cdef("size_t strlen(const char*);")
clib = ffi.dlopen(None)
length = clib.strlen("String to be evaluated.")
# prints: 23
print("{}".format(length))
```

ctypes

ctypes is the de facto library for interfacing with C/C++ from CPython, and it provides not only full access to the native C interface of most major operating systems (e.g., kernel32 on Windows, or libc on *nix), but also provides support for loading and interfacing with dynamic libraries, such as DLLs or shared objects at runtime. It does bring along with it a whole host of types for interacting with system APIs, and allows you to rather easily define your own complex types, such as structs and unions, and allows you to modify things such as padding and alignment, if needed. It can be a bit crufty to use, but in conjunction with the struct module, you are essentially provided full control over how your data types get translated into something usable by a pure C(++) method.

Struct Equivalents

MyStruct.h

```
struct my_struct {
    int a;
    int b;
};
```

MyStruct.py

```
import ctypes
class my_struct(ctypes.Structure):
    _fields_ = [("a", c_int),
                ("b", c_int)]
```

SWIG

SWIG, though not strictly Python focused (it supports a large number of scripting languages), is a tool for generating bindings for interpreted languages from C/C++ header files. It is extremely simple to use: the consumer simply needs

to define an interface file (detailed in the tutorial and documentations), include the requisite C/C++ headers, and run the build tool against them. While it does have some limits, (it currently seems to have issues with a small subset of newer C++ features, and getting template-heavy code to work can be a bit verbose), it provides a great deal of power and exposes lots of features to Python with little effort. Additionally, you can easily extend the bindings SWIG creates (in the interface file) to overload operators and built-in methods, effectively re- cast C++ exceptions to be catchable by Python, etc.

Example: Overloading __repr__

MyClass.h

```
#include <string>
class MyClass {
private:
    std::string name;
public:
    std::string getName();
};
```

myclass.i

```
%include "string.i"

%module myclass
%{
#include <string>
#include "MyClass.h"
%}

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

%include "MyClass.h"
```

Boost.Python

Boost.Python requires a bit more manual work to expose C++ object functionality, but it is capable of providing all the same features SWIG does and then some, to include providing wrappers to access PyObjects in C++, extracting SWIG- wrapper objects, and even embedding bits of Python into your C++ code.

CHAPTER 4

Python代码打包

本章主要聚焦于Python代码的部署。

Packaging Your Code

Package your code to share it with other developers. For example to share a library for other developers to use in their application, or for development tools like 'py.test'.

An advantage of this method of distribution is its well established ecosystem of tools such as PyPI and pip, which make it easy for other developers to download and install your package either for casual experiments, or as part of large, professional systems.

It is a well-established convention for Python code to be shared this way. If your code isn't packaged on PyPI, then it will be harder for other developers to find it, and to use it as part of their existing process. They will regard such projects with substantial suspicion of being either badly managed or abandoned.

The downside of distributing code like this is that it relies on the recipient understanding how to install the required version of Python, and being able and willing to use tools such as pip to install your code's other dependencies. This is fine when distributing to other developers, but makes this method unsuitable for distributing applications to end-users.

The Python Packaging Guide provides an extensive guide on creating and maintaining Python packages.

Alternatives to Packaging

To distribute applications to end-users, you should *freeze your application*.

On Linux, you may also want to consider *creating a Linux distro package* (e.g. a .deb file for Debian or Ubuntu.)

For Python Developers

If you're writing an open source Python module, PyPI , more properly known as *The Cheeseshop*, is the place to host it.

Pip vs. easy_install

Use pip. More details here

Personal PyPI

If you want to install packages from a source other than PyPI, (say, if your packages are *proprietary*), you can do it by hosting a simple http server, running from the directory which holds those packages which need to be installed.

Showing an example is always beneficial

For example, if you want to install a package called `MyPackage.tar.gz`, and assuming this is your directory structure:

- **archive**
  - **MyPackage**
    - MyPackage.tar.gz

Go to your command prompt and type:

```
$ cd archive
$ python -m SimpleHTTPServer 9000
```

This runs a simple http server running on port 9000 and will list all packages (like **MyPackage**). Now you can install **MyPackage** using any Python package installer. Using Pip, you would do it like:

```
$ pip install --extra-index-url=http://127.0.0.1:9000/ MyPackage
```

Having a folder with the same name as the package name is **crucial** here. I got fooled by that, one time. But if you feel that creating a folder called `MyPackage` and keeping `MyPackage.tar.gz` inside that, is *redundant*, you can still install MyPackage using:

```
$ pip install  http://127.0.0.1:9000/MyPackage.tar.gz
```

pypiserver

Pypiserver is a minimal PyPI compatible server. It can be used to serve a set of packages to easy_install or pip. It includes helpful features like an administrative command (`-U`) which will update all its packages to their latest versions found on PyPI.

S3-Hosted PyPi

One simple option for a personal PyPi server is to use Amazon S3. A prerequisite for this is that you have an Amazon AWS account with an S3 bucket.

1. **Install all your requirements from PyPi or another source**
2. **Install pip2pi**
   - `pip install git+https://github.com/wolever/pip2pi.git`
3. **Follow pip2pi README for pip2tgz and dir2pi commands**
   - `pip2tgz packages/ YourPackage` (or `pip2tgz packages/ -r requirements.txt`)
   - `dir2pi packages/`
4. **Upload the new files**
   - Use a client like Cyberduck to sync the entire `packages` folder to your s3 bucket
   - Make sure you upload `packages/simple/index.html` as well as all new files and directories

5. **Fix new file permissions**

- By default, when you upload new files to the S3 bucket, they will have the wrong permissions set.
- Use the Amazon web console to set the READ permission of the files to EVERYONE.
- If you get HTTP 403 when trying to install a package, make sure you've set the permissions correctly.

6. **All done**

- You can now install your package with `pip install --index-url=http://your-s3-bucket/packages/simple YourPackage`

For Linux Distributions

Creating a Linux distro package is arguably the "right way" to distribute code on Linux.

Because a distribution package doesn't include the Python interpreter, it makes the download and install about 2MB smaller than *freezing your application*.

Also, if a distribution releases a new security update for Python, then your application will automatically start using that new version of Python.

The bdist_rpm command makes producing an RPM file for use by distributions like Red Hat or SuSE is trivially easy.

However, creating and maintaining the different configurations required for each distribution's format (e.g. .deb for Debian/Ubuntu, .rpm for Red Hat/Fedora, etc) is a fair amount of work. If your code is an application that you plan to distribute on other platforms, then you'll also have to create and maintain the separate config required to freeze your application for Windows and OSX. It would be much less work to simply create and maintain a single config for one of the cross platform *freezing tools*, which will produce stand-alone executables for all distributions of Linux, as well as Windows and OSX.

Creating a distribution package is also problematic if your code is for a version of Python that isn't currently supported by a distribution. Having to tell *some versions* of Ubuntu end-users that they need to add the 'dead-snakes' PPA using *sudo apt-repository* commands before they can install your .deb file makes for an extremely hostile user experience. Not only that, but you'd have to maintain a custom equivalent of these instructions for every distribution, and worse, have your users read, understand, and act on them.

Having said all that, here's how to do it:

- Fedora
- Debian and Ubuntu
- Arch

Useful Tools

- fpm
- alien
- dh-virtualenv (for APT/DEB omnibus packaging)

Freezing Your Code

"Freezing" your code is creating a single-file executable file to distribute to end-users, that contains all of your application code as well as the Python interpreter.

Applications such as 'Dropbox', 'Eve Online', 'Civilization IV', and BitTorrent clients do this.

The advantage of distributing this way is that your application will "just work", even if the user doesn't already have the required version of Python (or any) installed. On Windows, and even on many Linux distributions and OS X, the right version of Python will not already be installed.

Besides, end-user software should always be in an executable format. Files ending in `.py` are for software engineers and system administrators.

One disadvantage of freezing is that it will increase the size of your distribution by about 2–12MB. Also, you will be responsible for shipping updated versions of your application when security vulnerabilities to Python are patched.

Alternatives to Freezing

Packaging your code is for distributing libraries or tools to other developers.

On Linux, an alternative to freezing is to *create a Linux distro package* (e.g. .deb files for Debian or Ubuntu, or .rpm files for Red Hat and SuSE.)

待处理

Fill in "Freezing Your Code" stub

Comparison of Freezing Tools

Solutions and platforms/features supported:

Solution	Windows	Linux	OS X	Python 3	License	One-file mode	Zipfile import	Eggs	pkg_resources support
bbFreeze	yes	yes	yes	no	MIT	no	yes	yes	yes
py2exe	yes	no	no	yes	MIT	yes	yes	no	no
pyInstaller	yes	yes	yes	yes	GPL	yes	no	yes	no
cx_Freeze	yes	yes	yes	yes	PSF	no	yes	yes	no
py2app	no	no	yes	yes	MIT	no	yes	yes	yes

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

注解： All solutions need MS Visual C++ dll to be installed on target machine, except py2app. Only Pyinstaller makes self-executable exe that bundles the dll when passing `--onefile` to `Configure.py`.

Windows

bbFreeze

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

待处理

Write steps for most basic .exe

py2exe

Prerequisite is to install *Python on Windows*.

1. Download and install http://sourceforge.net/projects/py2exe/files/py2exe/
2. Write `setup.py` (List of configuration options):

```
from distutils.core import setup
import py2exe

setup(
    windows=[{'script': 'foobar.py'}],
)
```

3. (Optionally) include icon
4. (Optionally) one-file mode
5. Generate `.exe` into `dist` directory:

```
$ python setup.py py2exe
```

6. Provide the Microsoft Visual C runtime DLL. Two options: globally install dll on target machine or distribute dll alongside with .exe.

PyInstaller

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

- Most basic tutorial
- Manual

OS X

py2app

PyInstaller

PyInstaller can be used to build Unix executables and windowed apps on Mac OS X 10.6 (Snow Leopard) or newer.

To install PyInstaller, use pip:

```
$ pip install pyinstaller
```

To create a standard Unix executable, from say `script.py`, use:

```
$ pyinstaller script.py
```

This creates,

- a `script.spec` file, analogous to a `make` file
- a `build` folder, that holds some log files

- a `dist` folder, that holds the main executable `script`, and some dependent Python libraries,

all in the same folder as `script.py`. PyInstaller puts all the Python libraries used in `script.py` into the `dist` folder, so when distributing the executable, distribute the whole `dist` folder.

The `script.spec` file can be edited to customise the build, with options such as

- bundling data files with the executable
- including run-time libraries (`.dll` or `.so` files) that PyInstaller can't infer automatically
- adding Python run-time options to the executable,

Now `script.spec` can be run with `pyinstaller` (instead of using `script.py` again):

```
$ pyinstaller script.spec
```

To create a standalone windowed OS X application, use the `--windowed` option

```
$ pyinstaller --windowed script.spec
```

This creates a `script.app` in the `dist` folder. Make sure to use GUI packages in your Python code, like PyQt or PySide, to control the graphical parts of the app.

There are several options in `script.spec` related to Mac OS X app bundles here. For example, to specify an icon for the app, use the `icon=\path\to\icon.icns` option.

Linux

bbFreeze

PyInstaller

CHAPTER 5

Python开发环境

本章主要聚焦于Python的开发环境以及编写Python代码工具的最佳实践。

Your Development Environment

Text Editors

Just about anything that can edit plain text will work for writing Python code, however, using a more powerful editor may make your life a bit easier.

Vim

Vim is a text editor which uses keyboard shortcuts for editing instead of menus or icons. There are a couple of plugins and settings for the Vim editor to aid Python development. If you only develop in Python, a good start is to set the default settings for indentation and line-wrapping to values compliant with **PEP 8**. In your home directory, open a file called `.vimrc` and add the following lines:

```
set textwidth=79  " lines longer than 79 columns will be broken
set shiftwidth=4  " operation >> indents 4 columns; << unindents 4 columns
set tabstop=4     " a hard TAB displays as 4 columns
set expandtab     " insert spaces when hitting TABs
set softtabstop=4 " insert/delete 4 spaces when hitting a TAB/BACKSPACE
set shiftround    " round indent to multiple of 'shiftwidth'
set autoindent    " align the new line indent with the previous line
```

With these settings, newlines are inserted after 79 characters and indentation is set to 4 spaces per tab. If you also use Vim for other languages, there is a handy plugin called indent, which handles indentation settings for Python source files.

There is also a handy syntax plugin called syntax featuring some improvements over the syntax file included in Vim 6.1.

These plugins supply you with a basic environment for developing in Python. To get the most out of Vim, you should continually check your code for syntax errors and PEP8 compliance. Luckily PEP8 and Pyflakes will do this for you. If your Vim is compiled with `+python` you can also utilize some very handy plugins to do these checks from within the editor.

For PEP8 checking and pyflakes, you can install vim-flake8. Now you can map the function `Flake8` to any hotkey or action you want in Vim. The plugin will display errors at the bottom of the screen, and provide an easy way to jump

to the corresponding line. It's very handy to call this function whenever you save a file. In order to do this, add the following line to your `.vimrc`:

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

If you are already using syntastic, you can set it to run Pyflakes on write and show errors and warnings in the quickfix window. An example configuration to do that which also shows status and warning messages in the statusbar would be:

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

Python-mode

Python-mode is a complex solution for working with Python code in Vim. It has:

- Asynchronous Python code checking (`pylint`, `pyflakes`, `pep8`, `mccabe`) in any combination
- Code refactoring and autocompletion with Rope
- Fast Python folding
- Virtualenv support
- Search through Python documentation and run Python code
- Auto PEP8 error fixes

And more.

SuperTab

SuperTab is a small Vim plugin that makes code completion more convenient by using `<Tab>` key or any other customized keys.

Emacs

Emacs is another powerful text editor. It is fully programmable (lisp), but it can be some work to wire up correctly. A good start if you're already an Emacs user is Python Programming in Emacs at EmacsWiki.

1. Emacs itself comes with a Python mode.

TextMate

> TextMate brings Apple's approach to operating systems into the world of text editors. By bridging UNIX underpinnings and GUI, TextMate cherry-picks the best of both worlds to the benefit of expert scripters and novice users alike.

Sublime Text

> Sublime Text is a sophisticated text editor for code, markup and prose. You'll love the slick user interface, extraordinary features and amazing performance.

Sublime Text has excellent support for editing Python code and uses Python for its plugin API. It also has a diverse variety of plugins, some of which allow for in-editor PEP8 checking and code "linting".

Atom

> Atom is a hackable text editor for the 21st century, built on atom-shell, and based on everything we love about our favorite editors.

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

IDEs

PyCharm / IntelliJ IDEA

PyCharm is developed by JetBrains, also known for IntelliJ IDEA. Both share the same code base and most of PyCharm's features can be brought to IntelliJ with the free Python Plug-In. There are two versions of PyCharm: Professional Edition (Free 30-day trial) and Community Edition (Apache 2.0 License) with fewer features.

Python (on Visual Studio Code)

Python for Visual Studio is an extension for the Visual Studio Code IDE. This is a free, light weight, open source IDE, with support for Mac, Windows, and Linux. Built using open source technologies such as Node.js and Python, with compelling features such as Intellisense (autocompletion), local and remote debugging, linting, and the like.

MIT licensed.

Enthought Canopy

Enthought Canopy is a Python IDE which is focused towards Scientists and Engineers as it provides pre installed libraries for data analysis.

Eclipse

The most popular Eclipse plugin for Python development is Aptana's PyDev.

Komodo IDE

Komodo IDE is developed by ActiveState and is a commercial IDE for Windows, Mac, and Linux. KomodoEdit is the open source alternative.

Spyder

Spyder is an IDE specifically geared toward working with scientific Python libraries (namely Scipy). It includes integration with pyflakes, pylint and rope.

Spyder is open-source (free), offers code completion, syntax highlighting, a class and function browser, and object inspection.

WingIDE

WingIDE is a Python specific IDE. It runs on Linux, Windows and Mac (as an X11 application, which frustrates some Mac users).

WingIDE offers code completion, syntax highlighting, source browser, graphical debugger and support for version control systems.

NINJA-IDE

NINJA-IDE (from the recursive acronym: "Ninja-IDE Is Not Just Another IDE") is a cross-platform IDE, specially designed to build Python applications, and runs on Linux/X11, Mac OS X and Windows desktop operating systems. Installers for these platforms can be downloaded from the website.

NINJA-IDE is open-source software (GPLv3 licence) and is developed in Python and Qt. The source files can be downloaded from GitHub.

Eric (The Eric Python IDE)

Eric is a full featured Python IDE offering sourcecode autocompletion, syntax highlighting, support for version control systems, python 3 support, integrated web browser, python shell, integrated debugger and a flexible plug-in system. Written in python, it is based on the Qt gui toolkit, integrating the Scintilla editor control. Eric is an open-source software project (GPLv3 licence) with more than ten years of active development.

Interpreter Tools

Virtual Environments

Virtual Environments provide a powerful way to isolate project package dependencies. This means that you can use packages particular to a Python project without installing them system wide and thus avoiding potential version conflicts.

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

pyenv

pyenv is a tool to allow multiple versions of the Python interpreter to be installed at the same time. This solves the problem of having different projects requiring different versions of Python. For example, it becomes very easy to install Python 2.7 for compatibility in one project, whilst still using Python 3.4 as the default interpreter. pyenv isn't just limited to the CPython versions - it will also install PyPy, anaconda, miniconda, stackless, jython, and ironpython interpreters.

pyenv works by filling a `shims` directory with fake versions of the Python interpreter (plus other tools like `pip` and `2to3`). When the system looks for a program named `python`, it looks inside the `shims` directory first, and uses the

fake version, which in turn passes the command on to pyenv. pyenv then works out which version of Python should be run based on environment variables, `.python-version` files, and the global default.

pyenv isn't a tool for managing virtual environments, but there is the plugin pyenv-virtualenv which automates the creation of different environments, and also makes it possible to use the existing pyenv tools to switch to different environments based on environment variables or `.python-version` files.

Other Tools

IDLE

IDLE is an integrated development environment that is part of Python standard library. It is completely written in Python and uses the Tkinter GUI toolkit. Though IDLE is not suited for full-blown development using Python, it is quite helpful to try out small Python snippets and experiment with different features in Python.

It provides the following features:

- Python Shell Window (interpreter)
- Multi window text editor that colorizes Python code
- Minimal debugging facility

IPython

IPython provides a rich toolkit to help you make the most out of using Python interactively. Its main components are:

- Powerful Python shells (terminal- and Qt-based).
- A web-based notebook with the same core features but support for rich media, text, code, mathematical expressions and inline plots.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into your own projects.
- Tools for high level and interactive parallel computing.

```
$ pip install ipython
```

To download and install IPython with all it's optional dependencies for the notebook, qtconsole, tests, and other functionalities

```
$ pip install ipython[all]
```

BPython

bpython is an alternative interface to the Python interpreter for Unix-like operating systems. It has the following features:

- In-line syntax highlighting.
- Readline-like autocomplete with suggestions displayed as you type.
- Expected parameter list for any Python function.
- "Rewind" function to pop the last line of code from memory and re-evaluate.
- Send entered code off to a pastebin.

- Save entered code to a file.
- Auto-indentation.
- Python 3 support.

```
$ pip install bpython
```

ptpython

ptpython is a REPL build on top of the prompt_toolkit library. It is considered to be an alternative to BPython. Features include:

- Syntax highlighting
- Autocompletion
- Multiline editing
- Emacs and VIM Mode
- Embedding REPL inside of your code
- Syntax Validation
- Tab pages
- Support for integrating with IPython's shell, by installing IPython `pip install ipython` and running `ptipython`.

```
$ pip install ptpython
```

Virtual Environments

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

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

virtualenv

virtualenv is a tool to create isolated Python environments. virtualenv creates a folder which contains all the necessary executables to use the packages that a Python project would need.

Install virtualenv via pip:

```
$ pip install virtualenv
```

Basic Usage

1. Create a virtual environment for a project:

```
$ cd my_project_folder
$ virtualenv venv
```

`virtualenv venv` will create a folder in the current directory which will contain the Python executable files, and a copy of the `pip` library which you can use to install other packages. The name of the virtual environment (in this case, it was `venv`) can be anything; omitting the name will place the files in the current directory instead.

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

You can also use the Python interpreter of your choice (like `python2.7`).

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

or change the interpreter globally with an env variable in `~/.bashrc`:

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

2. To begin using the virtual environment, it needs to be activated:

```
$ source venv/bin/activate
```

The name of the current virtual environment will now appear on the left of the prompt (e.g. `(venv)Your-Computer:your_project UserName$`) to let you know that it's active. From now on, any package that you install using pip will be placed in the `venv` folder, isolated from the global Python installation.

Install packages as usual, for example:

```
$ pip install requests
```

3. If you are done working in the virtual environment for the moment, you can deactivate it:

```
$ deactivate
```

This puts you back to the system's default Python interpreter with all its installed libraries.

To delete a virtual environment, just delete its folder. (In this case, it would be `rm -rf venv`.)

After a while, though, you might end up with a lot of virtual environments littered across your system, and its possible you'll forget their names or where they were placed.

Other Notes

Running `virtualenv` with the option `--no-site-packages` will not include the packages that are installed globally. This can be useful for keeping the package list clean in case it needs to be accessed later. [This is the default behavior for `virtualenv` 1.7 and later.]

In order to keep your environment consistent, it's a good idea to "freeze" the current state of the environment packages. To do this, run

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

This will create a `requirements.txt` file, which contains a simple list of all the packages in the current environment, and their respective versions. You can see the list of installed packages without the requirements format using "pip list". Later it will be easier for a different developer (or you, if you need to re-create the environment) to install the same packages using the same versions:

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

This can help ensure consistency across installations, across deployments, and across developers.

Lastly, remember to exclude the virtual environment folder from source control by adding it to the ignore list.

virtualenvwrapper

virtualenvwrapper provides a set of commands which makes working with virtual environments much more pleasant. It also places all your virtual environments in one place.

To install (make sure **virtualenv** is already installed):

```
$ pip install virtualenvwrapper
$ export WORKON_HOME=~/Envs
$ source /usr/local/bin/virtualenvwrapper.sh
```

(Full virtualenvwrapper install instructions.)

For Windows, you can use the virtualenvwrapper-win.

To install (make sure **virtualenv** is already installed):

```
$ pip install virtualenvwrapper-win
```

In Windows, the default path for WORKON_HOME is %USERPROFILE%Envs

Basic Usage

1. Create a virtual environment:

```
$ mkvirtualenv venv
```

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

2. Work on a virtual environment:

```
$ workon venv
```

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

```
$ mkproject myproject
```

virtualenvwrapper provides tab-completion on environment names. It really helps when you have a lot of environments and have trouble remembering their names.

`workon` also deactivates whatever environment you are currently in, so you can quickly switch between environments.

3. Deactivating is still the same:

```
$ deactivate
```

4. To delete:

```
$ rmvirtualenv venv
```

Other useful commands

`lsvirtualenv` List all of the environments.

`cdvirtualenv` Navigate into the directory of the currently activated virtual environment, so you can browse its `site-packages`, for example.

`cdsitepackages` Like the above, but directly into `site-packages` directory.

`lssitepackages` Shows contents of `site-packages` directory.

Full list of virtualenvwrapper commands.

virtualenv-burrito

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

autoenv

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

Install it on Mac OS X using `brew`:

```
$ brew install autoenv
```

And on Linux:

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

Further Configuration of Pip and Virtualenv

Requiring an active virtual environment for `pip`

By now it should be clear that using virtual environments is a great way to keep your development environment clean and keeping different projects' requirements separate.

When you start working on many different projects, it can be hard to remember to activate the related virtual environment when you come back to a specific project. As a result of this, it is very easy to install packages globally while thinking that you are actually installing the package for the virtual environment of the project. Over time this can result in a messy global package list.

In order to make sure that you install packages to your active virtual environment when you use `pip install`, consider adding the following line to your `~/.bashrc` file:

```
export PIP_REQUIRE_VIRTUALENV=true
```

After saving this change and sourcing the `~/.bashrc` file with `source ~/.bashrc`, pip will no longer let you install packages if you are not in a virtual environment. If you try to use `pip install` outside of a virtual environment pip will gently remind you that an activated virtual environment is needed to install packages.

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

You can also do this configuration by editing your `pip.conf` or `pip.ini` file. `pip.conf` is used by Unix and Mac OS X operating systems and it can be found at:

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

Similarly, the `pip.ini` file is used by Windows operating systems and it can be found at:

```
%HOME%\pip\pip.ini
```

If you don't have a `pip.conf` or `pip.ini` file at these locations, you can create a new file with the correct name for your operating system.

If you already have a configuration file, just add the following line under the `[global]` settings to require an active virtual environment:

```
require-virtualenv = true
```

If you did not have a configuration file, you will need to create a new one and add the following lines to this new file:

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

You will of course need to install some packages globally (usually ones that you use across different projects consistently) and this can be accomplished by adding the following to your `~/.bashrc` file:

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

After saving the changes and sourcing your `~/.bashrc` file you can now install packages globally by running `gpip install`. You can change the name of the function to anything you like, just keep in mind that you will have to use that name when trying to install packages globally with pip.

Caching packages for future use

Every developer has preferred libraries and when you are working on a lot of different projects, you are bound to have some overlap between the libraries that you use. For example, you may be using the `requests` library in a lot of different projects.

It is surely unnecessary to re-download the same packages/libraries each time you start working on a new project (and in a new virtual environment as a result). Fortunately, you can configure pip in such a way that it tries to reuse already installed packages.

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

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

You can set the path to anywhere you like (as long as you have write access). After adding this line, `source` your `.bashrc` (or `.bash_profile`) file and you will be all set.

Another way of doing the same configuration is via the `pip.conf` or `pip.ini` files, depending on your system. If you are on Windows, you can add the following line to your `pip.ini` file under `[global]` settings:

```
download-cache = %HOME%\pip\cache
```

Similarly, on UNIX systems you should simply add the following line to your `pip.conf` file under `[global]` settings:

```
download-cache = $HOME/.pip/cache
```

Even though you can use any path you like to store your cache, it is recommended that you create a new folder *in* the folder where your `pip.conf` or `pip.ini` file lives. If you don't trust yourself with all of this path voodoo, just use the values provided here and you will be fine.

CHAPTER 6

附加说明

本章内容比较单调，先介绍一些关于Python的背景，然后聚焦于接下来的内容。

Introduction

From the official Python website:

Python is a general-purpose, high-level programming language similar to Tcl, Perl, Ruby, Scheme, or Java. Some of its main key features include:

- **very clear, readable syntax**

  Python's philosophy focuses on readability, from code blocks delineated with significant whitespace to intuitive keywords in place of inscrutable punctuation.

- **extensive standard libraries and third party modules for virtually any task**

  Python is sometimes described with the words "batteries included" because of its extensive standard library, which includes modules for regular expressions, file IO, fraction handling, object serialization, and much more.

  Additionally, the Python Package Index is available for users to submit their packages for widespread use, similar to Perl's CPAN. There is a thriving community of very powerful Python frameworks and tools like the Django web framework and the NumPy set of math routines.

- **integration with other systems**

  Python can integrate with Java libraries, enabling it to be used with the rich Java environment that corporate programmers are used to. It can also be extended by C or C++ modules when speed is of the essence.

- **ubiquity on computers**

  Python is available on Windows, *nix, and Mac. It runs wherever the Java virtual machine runs, and the reference implementation CPython can help bring Python to wherever there is a working C compiler.

- **friendly community**

  Python has a vibrant and large *community* which maintains wikis, conferences, countless repositories, mailing lists, IRC channels, and so much more. Heck, the Python community is even helping to write this guide!

About This Guide

Purpose

The Hitchhiker's Guide to Python exists to provide both novice and expert Python developers a best practice handbook for the installation, configuration, and usage of Python on a daily basis.

By the Community

This guide is architected and maintained by Kenneth Reitz in an open fashion. This is a community-driven effort that serves one purpose: to serve the community.

For the Community

All contributions to the Guide are welcome, from Pythonistas of all levels. If you think there's a gap in what the Guide covers, fork the Guide on GitHub and submit a pull request.

Contributions are welcome from everyone, whether they're an old hand or a first-time Pythonista, and the authors to the Guide will gladly help if you have any questions about the appropriateness, completeness, or accuracy of a contribution.

To get started working on The Hitchhiker's Guide, see the Contribute page.

The Community

BDFL

Guido van Rossum, the creator of Python, is often referred to as the BDFL — the Benevolent Dictator For Life.

Python Software Foundation

The mission of the Python Software Foundation is to promote, protect, and advance the Python programming language, and to support and facilitate the growth of a diverse and international community of Python programmers.

Learn More about the PSF.

PEPs

PEPs are *Python Enhancement Proposals*. They describe changes to Python itself, or the standards around it.

There are three different types of PEPs (as defined by **PEP 1**):

Standards Describes a new feature or implementation.

Informational Describes a design issue, general guidelines, or information to the community.

Process Describes a process related to Python.

Notable PEPs

There are a few PEPs that could be considered required reading:

- **PEP 8: The Python Style Guide.** Read this. All of it. Follow it.
- **PEP 20: The Zen of Python.** A list of 19 statements that briefly explain the philosophy behind Python.

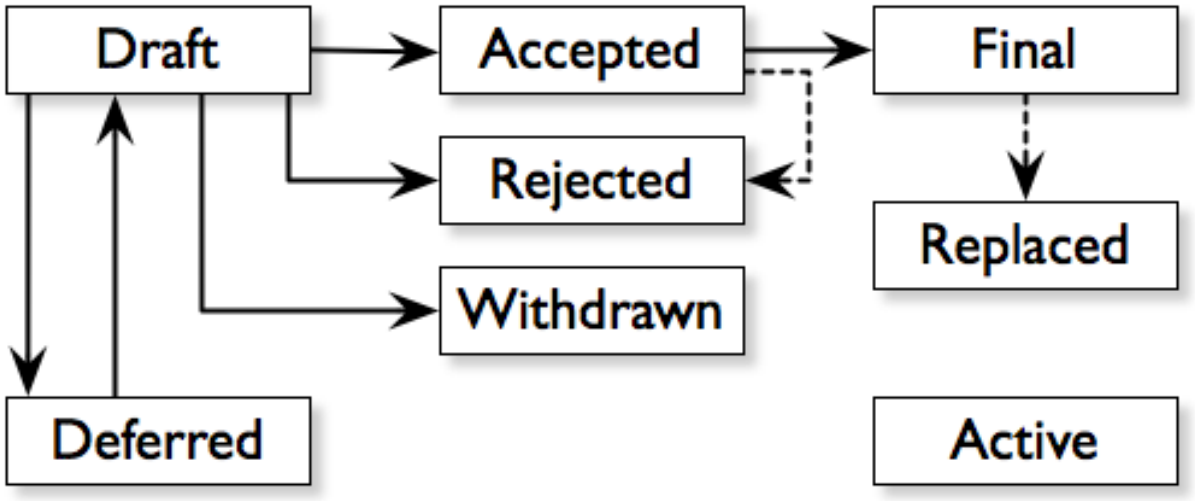
- **PEP 257: Docstring Conventions.** Gives guidelines for semantics and conventions associated with Python docstrings.

You can read more at The PEP Index.

Submitting a PEP

PEPs are peer-reviewed and accepted/rejected after much discussion. Anyone can write and submit a PEP for review.

Here's an overview of the PEP acceptance workflow:



Python Conferences

The major events for the Python community are developer conferences. The two most notable conferences are PyCon, which is held in the US, and its European sibling, EuroPython.

A comprehensive list of conferences is maintained at pycon.org.

Python User Groups

User Groups are where a bunch of Python developers meet to present or talk about Python topics of interest. A list of local user groups is maintained at the Python Software Foundation Wiki.

Learning Python

Beginner

The Python Tutorial

This is the official tutorial. It covers all the basics, and offers a tour of the language and the standard library. Recommended for those who need a quick-start guide to the language.

The Python Tutorial

Python for Beginners

thepythonguru.com is a tutorial focuses on beginner programmers. It covers many python concepts in depth. It also teaches you some advance constructs of python like lambda expression, regular expression. At last it finishes off with tutorial "How to access MySQL db using python"

Python for beginners

Learn Python Interactive Tutorial

Learnpython.org is an easy non-intimidating way to get introduced to Python. The website takes the same approach used on the popular Try Ruby website, it has an interactive Python interpreter built into the site that allows you to go through the lessons without having to install Python locally.

Learn Python

If you want a more traditional book, *Python For You and Me* is an excellent resource for learning all aspects of the language.

Python for You and Me

Online Python Tutor

Online Python Tutor gives you a visual step by step representation of how your program runs. Python Tutor helps people overcome a fundamental barrier to learning programming by understanding what happens as the computer executes each line of a program's source code.

Online Python Tutor

Invent Your Own Computer Games with Python

This beginner's book is for those with no programming experience at all. Each chapter has the source code to a small game, using these example programs to demonstrate programming concepts to give the reader an idea of what programs "look like".

Invent Your Own Computer Games with Python

Hacking Secret Ciphers with Python

This book teaches Python programming and basic cryptography for absolute beginners. The chapters provide the source code for various ciphers, as well as programs that can break them.

Hacking Secret Ciphers with Python

Learn Python the Hard Way

This is an excellent beginner programmer's guide to Python. It covers "hello world" from the console to the web.

Learn Python the Hard Way

Crash into Python

Also known as *Python for Programmers with 3 Hours*, this guide gives experienced developers from other languages a crash course on Python.

Crash into Python

Dive Into Python 3

Dive Into Python 3 is a good book for those ready to jump in to Python 3. It's a good read if you are moving from Python 2 to 3 or if you already have some experience programming in another language.

Dive Into Python 3

Think Python: How to Think Like a Computer Scientist

Think Python attempts to give an introduction to basic concepts in computer science through the use of the Python language. The focus was to create a book with plenty of exercises, minimal jargon and a section in each chapter devoted to the subject of debugging.

While exploring the various features available in the Python language the author weaves in various design patterns and best practices.

The book also includes several case studies which have the reader explore the topics discussed in the book in greater detail by applying those topics to real-world examples. Case studies include assignments in GUI and Markov Analysis.

Think Python

Python Koans

Python Koans is a port of Edgecase's Ruby Koans. It uses a test-driven approach, q.v. TEST DRIVEN DESIGN SECTION to provide an interactive tutorial teaching basic Python concepts. By fixing assertion statements that fail in a test script, this provides sequential steps to learning Python.

For those used to languages and figuring out puzzles on their own, this can be a fun, attractive option. For those new to Python and programming, having an additional resource or reference will be helpful.

Python Koans

More information about test driven development can be found at these resources:

Test Driven Development

A Byte of Python

A free introductory book that teaches Python at the beginner level, it assumes no previous programming experience.

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

Learn to Program in Python with Codeacademy

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

Learn to Program in Python with Codeacademy

Intermediate

Effective Python

This book contains 59 specific ways to improve writing Pythonic code. At 227 pages, it is a very brief overview of some of the most commons adapations programmers need to make to become efficient intermediate level Python programmers.

Effective Python

Advanced

Pro Python

This book is for intermediate to advanced Python programmers who are looking to understand how and why Python works the way it does and how they can take their code to the next level.

Pro Python

Expert Python Programming

Expert Python Programming deals with best practices in programming Python and is focused on the more advanced crowd.

It starts with topics like decorators (with caching, proxy, and context manager case-studies), method resolution order, using super() and meta-programming, and general **PEP 8** best practices.

It has a detailed, multi-chapter case study on writing and releasing a package and eventually an application, including a chapter on using zc.buildout. Later chapters detail best practices such as writing documentation, test-driven development, version control, optimization and profiling.

Expert Python Programming

A Guide to Python's Magic Methods

This is a collection of blog posts by Rafe Kettler which explain 'magic methods' in Python. Magic methods are surrounded by double underscores (i.e. __init__) and can make classes and objects behave in different and magical ways.

A Guide to Python's Magic Methods

注解： The Rafekettler.com is currently down, you can go to their Github version directly. Here you can find a PDF version: A Guide to Python's Magic Methods (repo on GitHub)

For Engineers and Scientists

A Primer on Scientific Programming with Python

A Primer on Scientific Programming with Python, written by Hans Petter Langtangen, mainly covers Python's usage in the scientific field. In the book, examples are chosen from mathematics and the natural sciences.

A Primer on Scientific Programming with Python

Numerical Methods in Engineering with Python

Numerical Methods in Engineering with Python, written by Jaan Kiusalaas, puts the emphasis on numerical methods and how to implement them in Python.

Numerical Methods in Engineering with Python

Miscellaneous topics

Problem Solving with Algorithms and Data Structures

Problem Solving with Algorithms and Data Structures covers a range of data structures and algorithms. All concepts are illustrated with Python code along with interactive samples that can be run directly in the browser.

Problem Solving with Algorithms and Data Structures

Programming Collective Intelligence

Programming Collective Intelligence introduces a wide array of basic machine learning and data mining methods. The exposition is not very mathematically formal, but rather focuses on explaining the underlying intuition and shows how to implement the algorithms in Python.

Programming Collective Intelligence

Transforming Code into Beautiful, Idiomatic Python

Transforming Code into Beautiful, Idiomatic Python is a video by Raymond Hettinger. Learn to take better advantage of Python's best features and improve existing code through a series of code transformations, "When you see this, do that instead."

Transforming Code into Beautiful, Idiomatic Python

Fullstack Python

Fullstack Python offers a complete top-to-bottom resource for web development using Python.

From setting up the webserver, to designing the front-end, choosing a database, optimizing/scaling, etc.

As the name suggests, it covers everything you need to build and run a complete web app from scratch.

Fullstack Python

References

Python in a Nutshell

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

Python in a Nutshell

The Python Language Reference

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

The Python Language Reference

Python Essential Reference

Python Essential Reference, written by David Beazley, is the definitive reference guide to Python. It concisely explains both the core language and the most essential parts of the standard library. It covers Python 3 and 2.6 versions.

Python Essential Reference

Python Pocket Reference

Python Pocket Reference, written by Mark Lutz, is an easy to use reference to the core language, with descriptions of commonly used modules and toolkits. It covers Python 3 and 2.6 versions.

Python Pocket Reference

Python Cookbook

Python Cookbook, written by David Beazley and Brian K. Jones, is packed with practical recipes. This book covers the core python language as well as tasks common to a wide variety of application domains.

Python Cookbook

Writing Idiomatic Python

"Writing Idiomatic Python", written by Jeff Knupp, contains the most common and important Python idioms in a format that maximizes identification and understanding. Each idiom is presented as a recommendation of a way to write some commonly used piece of code, followed by an explanation of why the idiom is important. It also contains two code samples for each idiom: the "Harmful" way to write it and the "Idiomatic" way.

For Python 2.7.3+

For Python 3.3+

Documentation

Official Documentation

The official Python Language and Library documentation can be found here:

- Python 2.x
- Python 3.x

Read the Docs

Read the Docs is a popular community project that hosts documentation for open source software. It holds documentation for many Python modules, both popular and exotic.

Read the Docs

pydoc

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

```
$ pydoc time
```

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

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

News

Planet Python

This is an aggregate of Python news from a growing number of developers.

Planet Python

/r/python

/r/python is the Reddit Python community where users contribute and vote on Python-related news.

/r/python

Pycoder's Weekly

Pycoder's Weekly is a free weekly Python newsletter for Python developers by Python developers (Projects, Articles, News, and Jobs).

Pycoder's Weekly

Python Weekly

Python Weekly is a free weekly newsletter featuring curated news, articles, new releases, jobs, etc. related to Python.

Python Weekly

Python News

Python News is the news section in the official Python web site (www.python.org). It briefly highlights the news from the Python community.

Python News

Import Python Weekly

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

Import Python Weekly Newsletter

Awesome Python Newsletter

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

Awesome Python Newsletter

注解: 所有的全音阶和半音音阶中定义的注意事项已经有意从附加说明这个名单中排除。除了此说明。（说实话，我也不太明白什么意思，字面翻译-_-||）

贡献说明以及法律信息（给那些感兴趣的人）。

Contribute

Python-guide is under active development, and contributors are welcome.

If you have a feature request, suggestion, or bug report, please open a new issue on GitHub. To submit patches, please send a pull request on GitHub. Once your changes get merged back in, you'll automatically be added to the Contributors List.

Style Guide

For all contributions, please follow the *The Guide Style Guide*.

Todo List

If you'd like to contribute, there's plenty to do. Here's a short todo list.

- 创建 "使用这个" vs "可选的是..." 的推荐介绍

待处理

Write about Blueprint

(原始记录 见 /home/docs/checkouts/readthedocs.org/user_builds/pyguide/checkouts/latest/docs/scenarios/admin.rst，第 369 行)

待处理

Fill in "Freezing Your Code" stub

(原始记录 见 /home/docs/checkouts/readthedocs.org/user_builds/pyguide/checkouts/latest/docs/shipping/freezing.rst，第 37 行)

待处理

Write steps for most basic .exe

(原始记录 见 /home/docs/checkouts/readthedocs.org/user_builds/pyguide/checkouts/latest/docs/shipping/freezing.rst，第 73 行)

待处理

增加上述每个项目里代码的典型示例。解释为什么这是优秀的代码。可以使用一些较为复杂的示例。

(原始记录 见 /home/docs/checkouts/readthedocs.org/user_builds/pyguide/checkouts/latest/docs/writing/reading.rst，第 35 行)

待处理

讲述一些可以快速确定数据结构、算法并确定代码实现什么功能的技术。

(原始记录 见 /home/docs/checkouts/readthedocs.org/user_builds/pyguide/checkouts/latest/docs/writing/reading.rst，第 37 行)

License

The Guide is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license.

The Guide Style Guide

As with all documentation, having a consistent format helps make the document more understandable. In order to make The Guide easier to digest, all contributions should fit within the rules of this style guide where appropriate.

The Guide is written as *reStructuredText*.

注解： 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

---

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

Relevancy

Strive to keep any contributions relevant to the *purpose of The Guide*.

- Avoid including too much information on subjects that don’t directly relate to Python development.

- Prefer to link to other sources if the information is already out there. Be sure to describe what and why you are linking.
- Cite references where needed.
- If a subject isn't directly relevant to Python, but useful in conjunction with Python (e.g., Git, GitHub, Databases), reference by linking to useful resources, and describe why it's useful to Python.
- When in doubt, ask.

Headings

Use the following styles for headings.

Chapter title:

```
#########
Chapter 1
#########
```

Page title:

```
====================
Time is an Illusion
====================
```

Section headings:

```
Lunchtime Doubly So
-------------------
```

Sub section headings:

```
Very Deep
~~~~~~~~~
```

Prose

Wrap text lines at 78 characters. Where necessary, lines may exceed 78 characters, especially if wrapping would make the source text more difficult to read.

Use of the serial comma (also known as the Oxford comma) is 100% non-optional. Any attempt to submit content with a missing serial comma will result in permanent banishment from this project, due to complete and total lack of taste.

Banishment? Is this a joke? Hopefully we will never have to find out.

Code Examples

Wrap all code examples at 70 characters to avoid horizontal scrollbars.

Command line examples:

```
.. code-block:: console

    $ run command --help
    $ ls ..
```

Be sure to include the `$` prefix before each line.

Python interpreter examples:

```
Label the example::

.. code-block:: python

    >>> import this
```

Python examples:

```
Descriptive title::

.. code-block:: python

    def get_answer():
        return 42
```

Externally Linking

- Prefer labels for well known subjects (ex: proper nouns) when linking:

```
    Sphinx_ is used to document Python.

    .. _Sphinx: http://sphinx.pocoo.org
```

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

```
    Read the `Sphinx Tutorial <http://sphinx.pocoo.org/tutorial.html>`_
```

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

Linking to Sections in The Guide

To cross-reference other parts of this documentation, use the :ref: keyword and labels.

To make reference labels more clear and unique, always add a `-ref` suffix:

```
.. _some-section-ref:

Some Section
------------
```

Notes and Warnings

Make use of the appropriate admonitions directives when making notes.

Notes:

```
.. note::
    The Hitchhiker's Guide to the Galaxy has a few things to say
    on the subject of towels. A towel, it says, is about the most
    massively useful thing an interstellar hitch hiker can have.
```

Warnings:

```
.. warning:: DON'T PANIC
```

TODOs

Please mark any incomplete areas of The Guide with a todo directive. To avoid cluttering the *Todo List*, use a single `todo` for stub documents or large incomplete sections.

```
.. todo::
    Learn the Ultimate Answer to the Ultimate Question
    of Life, The Universe, and Everything
```

P