

Python最佳实践指南 中文版

书栈(BookStack.CN)

目 录

致谢

介绍

Python入门

[选择一个 Python 解释器 \(3 vs 2\)](#)

[正确地安装 Python](#)

[在Mac OS X上安装Python 3](#)

[在Windows上安装Python 3](#)

[在Linux上安装Python 3](#)

[在Mac OS X上安装Python 2](#)

[在Windows上安装Python 2](#)

[在Linux上安装Python 2](#)

[Pipenv & 虚拟环境](#)

Python 开发环境

[您的开发环境](#)

[Pipenv & 虚拟环境](#)

[Pip和Virtualenv的更多配置](#)

写出优雅的Python代码

[结构化您的工程](#)

[代码风格](#)

[阅读好的代码](#)

[文档](#)

[测试您的代码](#)

[日志 \(Logging\)](#)

[常见陷阱](#)

[选择一个许可](#)

Python应用的场景指南

[网络应用](#)

[Web 应用 & 框架](#)

[HTML 抓取](#)

[命令行应用](#)

[GUI应用](#)

[数据库](#)

[网络](#)

[系统管理](#)

[持续集成](#)

[速度](#)

[科学应用](#)

[图像处理](#)

[数据序列化](#)

[XML解析](#)

[JSON](#)

[密码学](#)

[机器学习](#)

[与C/C++库交互](#)

[部署优雅的Python代码](#)

[打包您的代码](#)

[冻结（freezing）您的代码](#)

[额外关注](#)

[介绍](#)

[社区](#)

[学习Python](#)

[文档](#)

[新闻](#)

[贡献](#)

[许可证](#)

[风格指南指引](#)

致谢

当前文档 《Python最佳实践指南中文版》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-03-02。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/Python-Guide-CN>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Python最佳实践指南！

您好，地球人！欢迎来到Python最佳实践指南。

这是一份活着的、会呼吸的指南。 如果您有意一起贡献, [在GitHub fork 我!](#)

这份人工编写的指南旨在为Python初学者和专家提供一个关于Python安装、配置、和日常使用的最佳实践手册。

这份指南是 主观的，它与Python官方文档几乎，但不是完全 不同。您在这不会找到每个Python web框架的列表。相反，您会发现一份优秀的简明列表，包含有强烈推荐的选项。

注解

使用 **Python 3** 是 高度 优先于 Python 2。如果您发现自己 仍然 在生产环境中使用 Python 2，请考虑升级您的应用程序和基础设施。如果您正在使用 Python 3，恭喜您 — 您确实有很好的品味。—*Kenneth Reitz*

让我们开始吧！但首先，让我们确保您拥有这次旅行需要的"浴巾"。（译者注：towel 浴巾的梗引自著名科幻小说《银河系漫游指南》，大概是说先准备好不起眼但很重要的东西。）

Python入门

Python新手？让我们正确地设置您的Python环境：

- [选择一个 Python 解释器 \(3 vs 2\)](#)
- 正确地在您的系统上安装 Python
- 正确地安装 Python
- 在Mac OS X上安装Python 3
- 在Windows上安装Python 3
- 在Linux上安装Python 3
- 在Mac OS X上安装Python 2
- 在Windows上安装Python 2
- 在Linux上安装Python 2
- 借助 Pipenv 使用虚拟环境：
- [Pipenv & 虚拟环境](#)

Python 开发环境

这部分指南关注 Python 开发环境，以及用于编写 Python 代码的可用且最实用的工具。

- [您的开发环境](#)
- [Pipenv & 虚拟环境](#)
- [Pip和Virtualenv的更多配置](#)

写出优雅的Python代码

这部分指南关注编写Python代码的最佳实践。

- [结构化您的工程](#)
- [代码风格](#)
- [阅读好的代码](#)
- [文档](#)
- [测试您的代码](#)
- [日志 \(Logging\)](#)
- [常见陷阱](#)
- [选择一个许可](#)

Python应用的场景指南

这部分指南关注基于不同场景的工具和模块推荐。

- [网络应用](#)
- [Web 应用 & 框架](#)
- [HTML 抓取](#)
- [命令行应用](#)
- [GUI应用](#)
- [数据库](#)
- [网络](#)
- [系统管理](#)
- [持续集成](#)
- [速度](#)
- [科学应用](#)
- [图像处理](#)
- [数据序列化](#)
- [XML解析](#)
- [JSON](#)
- [密码学](#)
- [机器学习](#)
- [与C/C++库交互](#)

部署优雅的Python代码

这部分指南关注部署您的Python代码。

- [打包您的代码](#)
- [冻结 \(freezing\) 您的代码](#)

额外关注

这部分指南比较零散，先了解一些Python的背景知识，再关注下一步。

- [介绍](#)
 - [社区](#)
 - [学习Python](#)
 - [文档](#)
 - [新闻](#)
-

贡献注意点和法律信息如下（给感兴趣的同学）：

- [贡献](#)
- [许可证](#)
- [风格指南指引](#)

原文：<http://pythonguidecn.readthedocs.io/zh/latest/>

- 选择一个 Python 解释器 (3 vs 2)
- 正确地安装 Python
- 在Mac OS X上安装Python 3
- 在Windows上安装Python 3
- 在Linux上安装Python 3
- 在Mac OS X上安装Python 2
- 在Windows上安装Python 2
- 在Linux上安装Python 2
- Pipenv & 虚拟环境

选择一个 Python 解释器 (3 vs 2)



Python的现状 (2 vs 3)

当选择Python解释器的时候，一个首先要面对的问题是：“我应该选择Python 2还是Python 3？” 答案并不像人们想象的那么明显。

现状的基本要点如下：

- 如今大部分生产应用使用 Python 2.7。
- Python 3 已准备好用于生产应用的部署。
- Python 2.7 直到 2020 前只会得到必要的安全更新 [\[1\]](#)。
- “Python” 涵盖了 Python 3 和 Python 2。

建议

注解

使用 **Python 3** 是 高度 优先于 Python 2。如果您发现自己 仍然 在生产环境中使用 Python 2，请考虑升级您的应用程序和基础设施。如果您正在使用 Python 3，恭喜您 — 您确实有很好的品味。—*Kenneth Reitz*

那我直言不讳：

- 将 Python 3 用于新的 Python 应用程序。
- 如果您是第一次学习 Python，熟悉 Python 2.7 将是非常有用的，但学习 Python 3 更有用。
- 两者都学。它们都是 “Python”。
- 已经构建的软件通常依赖于 Python 2.7。
- 如果您正在编写一个新的开源 Python 库，最好同时为 Python 2 和 3 编写。若新库只支持 Python 3 会是一项政治声明，并将疏远您的许多用户。这不是一个问题 — 慢慢地，在未来三年内，这种情况会减少。

所以... 3 ?

如果您想选择一种Python的解释器，您又不是固执己见的人，我推荐您用最新的Python 3.x，因为每个版本都带来了新的改进了的标准库模块、安全性以及bug修复。进步就是进步。

鉴于此，如果您有一个强有力的理由只用Python 2，比如Python 3 没有足够的替代的Python 2的特有库，或者您（像我）非常喜而且受Python 2启发。

查看 [Can I Use Python 3?](#) 来看看是否有您依赖的软件阻止您用Python 3。

延伸阅读

写 [能够同时兼容Python 2.6, 2.7, 和Python 3的代码](#) 是可能的。这包括从简单到困难的各种难度，取决于您所写软件的类型；如果您是初学者，其实有更重要的东西要操心。请注意，Python 2.6是生命周期结束的上游，所以您不应该试着编写兼容2.6的代码，除非您被专门安排做这件事。

实现

当人们谈论起 *Python*，他们不仅是在说语言本身，还包括其CPython实现。*Python* 实际上是一个可以用许多不同的方式来实现的语言规范。

CPython

[CPython](#) 是Python的参考实现，用C编写。它把Python代码编译成中间态的字节码，然后由虚拟机解释。CPython为Python包和C扩展模块提供了最大限度的兼容。

如果您正在写开源的Python代码，并希望有尽可能广泛的用户，用CPython是最好的。使用依赖C扩展的包，CPython是您唯一的选择。

所有版本的Python语言都用C实现，因为CPython是参考实现。

PyPy

[PyPy](#) 是用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即时编译，支持多重后端 (C, CLI, JVM)。

PyPy旨在提高性能，同时保持最大兼容性（参考CPython的实现）。

如果您正在寻找提高您的Python代码性能的方法，值得试一试PyPy。在一套的基准测试下，它目前比CPython的速度快超过5倍。

PyPy支持Python 2.7。PyPy3 [2]，发布的Beta版，支持Python 3。

Jython

Jython 是一个将Python代码编译成Java字节码的实现，运行在JVM (Java Virtual Machine) 上。另外，它可以像是用Python模块一样，导入并使用任何Java类。

如果您需要与现有的Java代码库对接或者基于其他原因需要为JVM编写Python代码，那么Jython是最好的选择。

Jython现在支持到Python 2.7 [3]。

IronPython

IronPython 是一个针对 .NET 框架的Python实现。它可以用Python和.NET framework的库，也能将Python代码暴露给给.NET框架中的其他语言。

Python Tools for Visual Studio 直接集成了IronPython到Visual Studio开发环境中，使之成为Windows开发者的理想选择。

IronPython支持Python 2.7 [4]。

PythonNet

Python for .NET 是一个近乎无缝集成的，提供给本机已安装的Python .NET公共语言运行时 (CLR) 包。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。

PythonNet与Mono相结合使用，通过 .NET框架，能使Python在非windows系统上 (如OS X和Linux) 完成操作。它可以在除外IronPython的环境中无冲突运行。

PythonNet支持Python 2.3到2.7 [5]。

[1] | <https://www.python.org/dev/peps/pep-0373/#id2>

[2] | <http://pypy.org/compat.html>

[3] | <https://hg.python.org/jython/file/412a8f9445f7/NEWS>

[4] | <http://ironpython.codeplex.com/releases/view/81726>

[5] | <http://pythontut.githup.io/readme.html>

原文: <http://pythonguidecn.readthedocs.io/zh/latest/starting/which-python.html>

正确地安装 Python



通常来说，您所使用的操作系统自带安装好的Python。

如此可以省却自己安装、配置 Python 的过程。话虽如此，我还是强烈建议各位，在正式开始 Python 应用开发前，安装接下来教程中所介绍的工具和库。特别要提到如 `setuptools`, `pip` 和 `virtualenv` 这样的工具——它们将简化安装和使用Python第三方库的流程。

注解

使用 **Python 3** 的优先级高于Python2。如果您发现自己 仍然 在生产环境中使用 Python 2，请考虑升级您的应用程序和基础设施。如果您正在使用 Python 3，恭喜您 — 您确实有很好的品味。—*Kenneth Reitz*

安装指南

这份指南重温了 [Python](#) 作为开发工具的正确安装方法，也包括 `setuptools`, `pip` 和 `virtualenv` 等工具的安装。

Python 3 安装指南

- [MacOS 上的 Python 3.](#)
- [Microsoft Windows 上的 Python 3.](#)
- [Linux 上的 Python 3.](#)

传统 Python 2 安装指南

- [MacOS 上的 Python 2.](#)
- [Microsoft Windows 上的 Python 2.](#)
- [Linux 上的 Python 2.](#)

原文: <http://pythonguidecn.readthedocs.io/zh/latest/starting/installation.html>

在Mac OS X上安装Python 3



最新版本的Mac OS X, High Sierra, 自带Python 2.7。

您不必安装和配置即可直接使用Python 2。本教程用来说明Python 3的安装。

OS X自带的Python版本更适合用于学习而不是开发。因为版本与Python官网发布的 [官方最新稳定版本](#) 相比可能已经过时。

现在就开始吧！

跟着我一起安装真实版本的Python吧。

在正式安装之前，应先安装C编译器。最快的方式是运行 `xcode-select -install` 来安装Xcode命令行工具。您也可以从Mac应用商店下载完全版的 [XCode](#)，或者更轻巧的 [OSX-GCC-Installer](#)。

注解

如果已经安装了XCode，请不要再安装 OSX-GCC-Installer。两者结合可能会引发难以诊断的问题。

注解

执行XCode的全新安装完成后，须在终端执行下述命令 `xcode-select -install` 来安装命令行工具。

尽管OS X系统附带了大量UNIX工具，熟悉Linux系统的人员使用时会发现缺少一个重要的组件—合适的包管理工具，[Homebrew](#) 正好填补了这个空缺。

安装 Homebrew 只需打开 [终端](#) 或个人常用的终端模拟器并运行：

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

运行这段脚本将列出它会引起的改变，并在安装开始前提示您。安装完成Homebrew后，需将其所在路径插入到 [PATH](#) 环境变量的最前面，即在您所登录用户的 [~/.profile](#) 文件末尾加上这一行：

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

接下来可以开始安装Python 3：

```
1. $ brew install python
```

这将持续几分钟。

Pip

Homebrew 会为您安装 [pip3](#)。

[pip3](#) 是Homebrew版Python 3的 [pip](#) 的别名。

使用Python 3

这个时候，在您系统上可能Python 2.7也是可用的。可能 [Homebrew 版本的Python 2](#)和Python 3都安装了。

```
1. $ python
```

将打开通过HomeBrew安装的Python解释器。

```
1. $ python2
```

将会打开使用Homebrew安装的Python 2解释器（如果有）。

```
1. $ python3
```

将会打开使用Homebrew安装的Python 3解释器（如果有）。

如果Homebrew版的Python 2安装了， [pip2](#) 指向Python 2。如果Homebrew版的Python 3安装了， [pip](#) 指向Python 3。

本指南的其余部分假定 [python](#) 指 Python 3。

```
1. # 我安装Python 3了吗?  
2. $ python --version  
3. Python 3.6.4 # Success!  
4. # If you still see 2.7 ensure in PATH /usr/local/bin/ takes precedence over /usr/bin/
```

Pipenv & 虚拟环境

下一步安装 Pipenv，然后就可以安装依赖关系并管理虚拟环境。

虚拟环境工具通过为不同项目创建专属的 Python 虚拟环境，以实现其依赖的库独立保存在不同的路径。这解决了“项目X依赖于 1.x 版本，但项目 Y 需要 4.x”的难题，并且维持全局的 site-packages 目录干净、易管理。

举个例子，通过这个工具可以实现依赖 Django 1.10 的项目与依赖 Django 1.8 的项目共存。

所以，向前！进入到 [Pipenv & 虚拟环境](#) 文档中！

该页是 [另一份指南](#) 的混合版本，可通过同一份许可获取。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/starting/install3/osx.html>



首先，遵照 [Chocolatey](#) 的安装指引。它是 Windows 7+ 的社区系统包管理器（很像Mac OSX上的 Homebrew）。

完成之后，安装Python 3会非常简单，因为Chocolatey将Python 3作为默认设置。

```
1. choco install python
```

一旦您运行了上述命令，您应该能够直接从控制台启动Python。（ Chocolatey非常棒，会自动将Python添加到您的系统路径中。）

SetupTools + Pip

[setupTools](#) 和 [pip](#)是两个最重要的第三方Python包。

安装完成后，您可以使用单个命令下载、安装和卸载任何兼容的Python应用包。还可以轻松地这种网络安装的方式加入到自己开发的Python应用中。

所有受支持的Python 3版本都包含pip，因此请确保它是最新的：

```
1. python -m pip install -U pip
```

Pipenv & 虚拟环境

下一步安装 Pipenv，然后就可以安装依赖关系并管理虚拟环境。

虚拟环境工具通过为不同项目创建专属的 Python 虚拟环境，以实现其依赖的库独立保存在不同的路径。这解决了“项目X依赖于 1.x 版本，但项目 Y 需要 4.x”的难题，并且维持全局的 site-packages 目录干净、易管理。

举个例子，通过这个工具可以实现依赖 Django 2.0 的项目与依赖 Django 1.8 的项目共存。

所以，向前！进入到 [Pipenv & 虚拟环境](#) 文档中！

该页是 [另一份指南](#) 的混合版本，可通过同一份许可获取。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/starting/install3/win.html>

在Linux上安装Python 3



这份文档描述了如何在Ubuntu Linux机器上安装Python 3.6。

想要获取已安装的Python 3版本号，可以通过终端运行命令：

```
1. $ python3 --version
```

如果您使用的是Ubuntu 16.10或更新，可以通过以下命令简单地安装Python 3.6：

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

如果您使用的是其他版本的Ubuntu（比如LTS发行版），我们推荐使用 [deadsnakes PPA](#) 来安装 Python 3.6：

```
1. $ sudo apt-get install software-properties-common  
2. $ sudo add-apt-repository ppa:deadsnakes/ppa  
3. $ sudo apt-get update  
4. $ sudo apt-get install python3.6
```

如果您使用的是其他Linux发行版，有可能已经预装了Python 3。如果没有，使用发行版的包管理器。比如，在Fedora上您可以使用 *dnf*：

```
1. $ sudo dnf install python3
```

注意，如果 `python3` 包的版本不够新，还有其他方式安装更新的版本，这取决于所在的发行版。比如在Fedora 25上安装 `python36` 包来获取Python 3.6。如果您是Fedora用户，您可能想阅读 [Fedora中可用的多Python版本](#)。

使用Python 3

这个时候，在您系统上可能Python 2.7也是可用的。

```
1. $ python
```

将打开Python 2解释器。

```
1. $ python3
```

将打开Python 3解释器。

SetupTools & Pip

[setupTools](#) 和 [pip](#)是最重要的两个Python第三方软件包。一旦安装了它们，就可以通过一条指令下载、安装和卸载可获取到的Python应用包，还可以轻松地将这种网络安装的方式加入到自己开发的Python应用中。

Python 2.7.9 以及之后版本(Python2 系列)，和Python 3.4以及之后版本均默认包含pip。

运行以下命令行代码检查pip是否已经安装：

```
1. $ command -v pip
```

[参考官方pip安装指南](#) 获取pip工具，并自动安装最新版本的setupTools。

注意，在某些Linux发行版（包括Ubuntu和Fedora）上，`pip` 用于Python 2的，而 `pip3` 用于Python 3。

```
1. $ command -v pip3
```

不过，使用虚拟环境（下面描述）就无需担心这个问题。

Pipenv & 虚拟环境

下一步安装 Pipenv，然后就可以安装依赖关系并管理虚拟环境。

虚拟环境工具通过为不同项目创建专属的 Python 虚拟环境，以实现其依赖的库独立保存在不同的路径。这解决了“项目X依赖于 1.x 版本，但项目 Y 需要 4.x”的难题，并且维持全局的 site-packages 目录干净、易管

理。

举个例子，通过这个工具可以实现依赖 Django 1.10 的项目与依赖 Django 1.8 的项目共存。

所以，向前！进入到 [Pipenv & 虚拟环境](#) 文档中！

该页是 [另一份指南](#) 的混合版本，可通过同一份许可获取。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/starting/install3/linux.html>

在Mac OS X上安装Python 2



注解

查看我们的 [在OS X上安装Python 3指南](#)。

最新版本的Mac OS X, High Sierra, 自带**Python 2.7**。

您不必安装和配置即可直接使用Python进行开发。话虽如此，我还是强烈建议各位，在正式开始Python应用开发前，安装接下来教程中所介绍的工具和库。特别应该安装**Setuptools**—它将简化安装和使用Python第三方库的流程。

OS X自带的Python版本更适合用于学习而不是开发。因为版本与Python官网发布的 [官方最新稳定版本](#) 相比可能已经过时。

现在就开始吧！

跟着我一起安装真实版本的Python吧。

在正式安装之前，应先安装C编译器。最快的方式是运行 `xcode-select -install` 来安装Xcode命令行工具。您也可以从Mac应用商店下载完全版的 [XCode](#)，或者更轻巧的 [OSX-GCC-Installer](#)。

注解

如果已经安装了XCode, 请不要再安装 OSX-GCC-Installer。两者结合可能会引发难以诊断的问题。

注解

执行XCode的全新安装完成后, 须在终端执行下述命令 `xcode-select --install` 来安装命令行工具。

尽管OS X系统附带了大量UNIX工具, 熟悉Linux系统的人员使用时会发现缺少一个重要的组件—合适的包管理工具, [Homebrew](#) 正好填补了这个空缺。

安装 Homebrew 只需打开 [终端](#) 或个人常用的终端模拟器并运行:

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

运行这段脚本将列出它会引起的改变, 并在安装开始前提示您。安装完成Homebrew后, 需将其所在路径插入到 `PATH` 环境变量的最前面, 即在您所登录用户的 `~/.profile` 文件末尾加上这一行:

```
1. export PATH="/usr/local/bin:/usr/local/sbin:$PATH"
```

接下来可以开始安装Python 2.7:

```
1. $ brew install python@2
```

因为 `python@2` 像一个“桶”, 我们需要再次更新我们的 `PATH`, 以指向我们的新安装:

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

Homebrew命名可执行文件 `python2`, 以便您仍然可以通过可执行文件“python”运行系统Python。

```
1. $ python -V # Homebrew安装的Python 3解释器 (如果安装了)  
2. $ python2 -V # Homebrew安装的Python 2解释器  
3. $ python3 -V # Homebrew安装的Python 3解释器 (如果安装了)
```

SetupTools & Pip

Homebrew会自动安装好SetupTools和 `pip`。SetupTools提供 `easy_install` 命令, 实现通过网络(通常Internet)下载和安装第三方Python包。还可以轻松地将这种网络安装的方式加入到自己开发的Python应用中。

`pip` 是一款方便安装和管理Python 包的工具, 在 [一些方面](#), 它更优于 `easy_install`, 故更推荐它。

```
1. $ pip2 -V # pip指向Homebrew安装的Python 2解释器  
2. $ pip -V # pip指向Homebrew安装的Python 3解释器 (如果安装了)
```

虚拟环境(Virtual Environment)

虚拟环境工具(`virtualenv`)通过为不同项目创建专属的Python虚拟环境, 以实现其依赖的库独立保存在不同的路

径。这解决了“项目X依赖包版本1.x，但项目Y依赖包版本为4.x”的难题，并且维持全局的site-packages目录干净、易管理。

举个例子，通过这个工具可以实现依赖Django 1.10的项目与依赖Django 1.8的项目共存。

进一步了解与使用请参考文档 [Virtual Environments](#)。

该页是 [另一份指南](#) 的混合版本，可通过同一份许可获取。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/starting/install/osx.html>



注解

查看我们的 [在Windows上安装Python 3指南](#).

首先，从官网下载 [最新版本](#)的Python 2.7。如果想要确保下载到最新版本，单击 [Python官网](#) 的Downloads > Windows 链接。

Windows版本是MSI文件格式，双击它即可开始安装。MSI文件格式允许Windows管理员使用标准工具自动进行安装流程。

Python将安装到内含版本号信息的路径，例如Python 2.7版本将被安装到 `C:\Python27\`，故多个版本的Python可以共存在一个系统里，不会有冲突。当然仅有一个默认的Python解释器，`PATH` 环境变量也不是自动修改的，开发人员可以控制要运行的Python版本。

把默认使用的Python版本路径加到 `PATH` 环境变量中，避免每次使用时都要冗余地写全Python解释器路径。假设安装路径是 `C:\Python27\`，将这段加入到 `PATH` 中：

```
1. C:\Python27\;C:\Python27\Scripts\
```

或在 `powershell` 中运行：

```
1. [Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\;C:\Python27\Scripts\", "User")
```

这也是安装过程中的一个选择。

其中第二个路径(`Scripts`)可接收已安装包的命令文件，添加这个路径很有益处。虽然以上步骤完成后，就可以开始正式使用Python了，但我还是强烈建议各位，在正式开始Python应用开发前，安装接下来教程中所介绍的工具和库。特别应该安装`Setuptools`—它将简化安装和使用Python第三方库的流程。

Setuptools + Pip

`setuptools` 和 `pip`是两个最重要的第三方Python包。

安装完成后，您可以使用单个命令下载、安装和卸载任何兼容的Python应用包。还可以轻松地这种网络安装的方式加入到自己开发的Python应用中。

Python 2.7.9和更高版本 (在Python 2系列上)，Python 3.4和更高版本默认包含`pip`。

要查看是否安装了`pip`，请打开命令提示符并运行：

```
1. $ command -v pip
```

要安装`pip`，请遵照官方的[pip安装指南](#) — 这将自动安装最新版本的`setuptools`。

虚拟环境

虚拟环境工具(通常是指“`virtualenv`”)通过为不同项目创建专属的Python虚拟环境，以实现其依赖的库独立保存在不同的路径。这解决了“项目X依赖包版本1.x，但项目Y依赖包版本为4.x”的难题，并且维持全局的`site-packages`目录干净、易管理。

举个例子，通过这个工具可以实现依赖Django 1.10的项目与依赖Django 1.8的项目共存。

进一步了解与使用请参考文档 [Virtual Environments](#)。

该页是 [另一份指南](#) 的混合版本，可通过同一份许可获取。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/starting/install/win.html>



注解

查看我们的 [在Linux上安装Python 3指南](#)。

最新版本的CentOS, Fedora, Red Hat 企业版 Linux (RHEL) 和 Ubuntu 自带 **Python 2.7**。

想要获取已安装的Python 2版本号，可以通过终端运行命令：

```
1. $ python2 --version
```

尽管如此，随着Python 3的流行，一些发行版，比如Fedora，不再预装Python 2。您可以使用发行版的包管理器来安装 `python2`：

```
1. $ sudo dnf install python2
```

您不必安装和配置即可直接使用Python进行开发。话虽如此，我还是强烈建议各位，在正式开始Python应用开发前，安装接下来教程中所介绍的工具和库。特别应该安装SetupTools和pip，它们将简化安装和使用Python第三方库的流程。

SetupTools & Pip

`setuptools` 和 `pip`是最重要的两个Python第三方软件包。一旦安装了它们，就可以通过一条指令下载、安装和卸载可获取到的Python应用包，还可以轻松地将这种网络安装的方式加入到自己开发的Python应用中。

Python 2.7.9 以及之后版本(Python2 系列)，和Python 3.4以及之后版本均默认包含pip。

运行以下命令行代码检查pip是否已经安装：

```
1. $ command -v pip
```

参考官方[pip安装指南](#) 获取pip工具，并自动安装最新版本的`setuptools`。

虚拟环境

虚拟环境工具(`virtualenv`)通过为不同项目创建专属的Python虚拟环境，以实现其依赖的库独立保存在不同的路径。这解决了“项目X依赖包版本1.x，但项目Y依赖包版本为4.x”的难题，并且维持全局的`site-packages`目录干净、易管理。

举个例子，通过这个工具可以实现依赖Django 1.10的项目与依赖Django 1.8的项目共存。

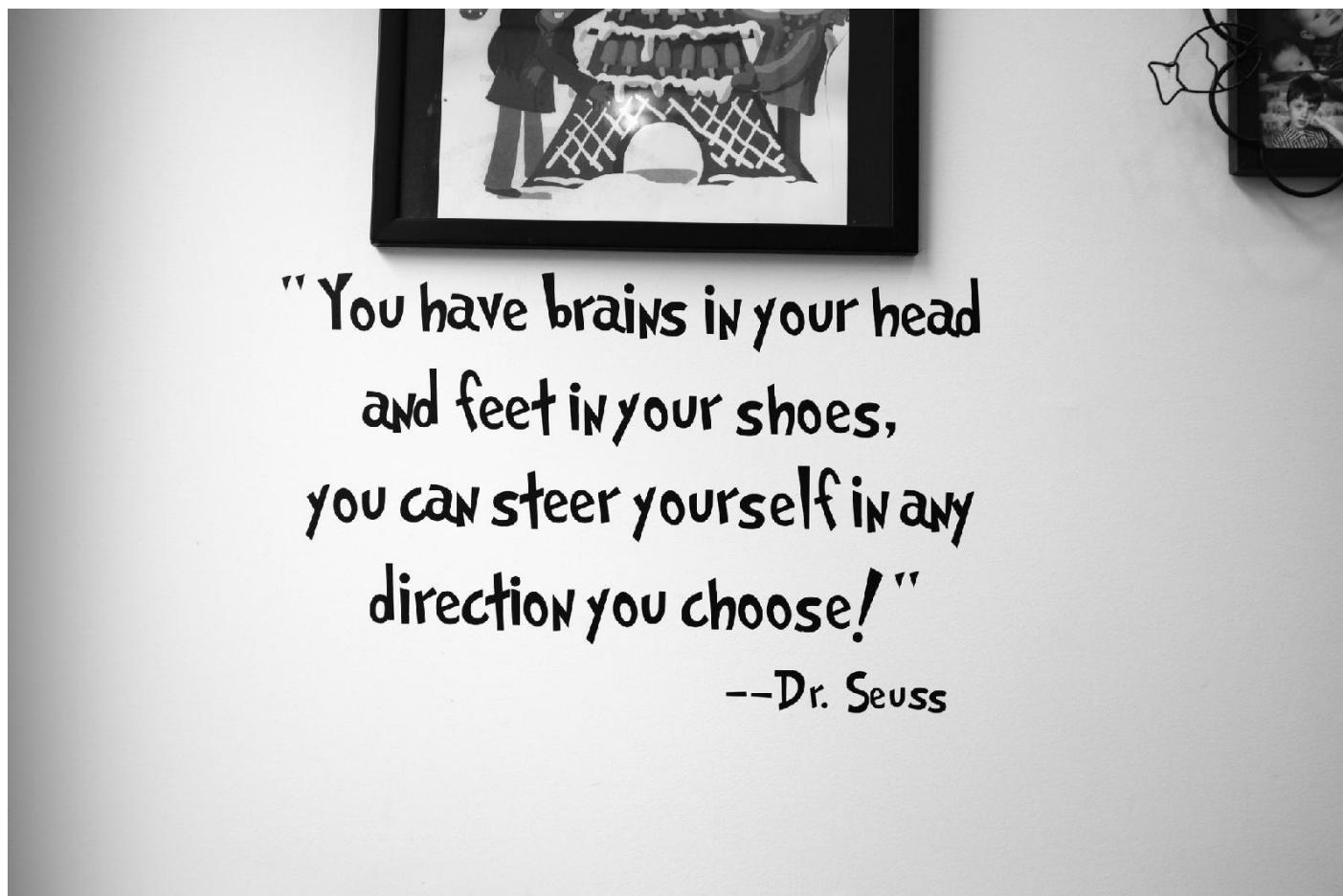
进一步了解与使用请参考文档 [虚拟环境](#)。

也可使用 `virtualenvwrapper` 更轻松地管理您的虚拟环境。

该页是 [另一份指南](#) 的混合版本，可通过同一份许可获取。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/starting/install/linux.html>

Pipenv & 虚拟环境



本教程将引导您完成安装和使用 Python 包。

它将向您展示如何安装和使用必要的工具，并就最佳做法做出强烈推荐。请记住，Python 用于许多不同的目的。准确地说，您希望如何管理依赖项可能会根据您如何决定发布软件而发生变化。这里提供的指导最直接适用于网络服务（包括 Web 应用程序）的开发和部署，但也非常适合管理任意项目的开发和测试环境。

注解

本指南是针对 Python 3 编写。但如果由于某种原因仍然使用 Python 2.7，这些指引应该能够正常工作。

确保您已经有了 Python 和 pip

在您进一步之前，请确保您有 Python，并且可从您的命令行中获得。您可以通过简单地运行以下命令来检查：

```
1. $ python --version
```

您应该得到像 [3.6.2](#) 之类的一些输出。如果没有 Python，请从 python.org 安装最新的 3.x 版本，或参考本指南的 [安装 Python](#) 一节。

注解

如果您是新手，您会得到如下错误：

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

这是因为此命令要在 `shell`（也称为 终端 或 控制台）中运行。有关使用操作系统的`shell` 并和 Python 进行交互的介绍，请参阅面向 Python 新手的 [入门教程](#)。

另外，您需要确保 `pip` 是可用的。您可以通过运行以下命令来检查：

```
1. $ pip --version
```

如果您使用 [python.org](#) 或 [Homebrew](#) 的安装程序来安装 Python，您应该已经有 `pip` 了。如果您使用的是 Linux，并使用操作系统的包管理器进行安装，则可能需要单独[安装 pip](#)。

安装 Pipenv

[Pipenv](#) 是 Python 项目的依赖管理器。如果您熟悉 Node.js 的 `npm` 或 Ruby 的 `bundler`，那么它们在思路上与这些工具类似。尽管 `pip` 可以安装 Python 包，但仍推荐使用 Pipenv，因为它是一种更高级的工具，可简化依赖关系管理的常见使用情况。

使用 `pip` 来安装 Pipenv：

```
1. $ pip install --user pipenv
```

注解

这进行了 [用户安装](#)，以防止破坏任何系统范围的包。如果安装后，`shell` 中没有 `pipenv`，则需要将 [用户基础目录](#) 的二进制文件目录添加到 `PATH` 中。

在 Linux 和 macOS 上，您可以通过运行 `python -m site --user-base` 找到用户基础目录，然后把 `bin` 加到目录末尾。比如，上述命令典型地会打印出 `~/.local/bin`（`~` 会扩展为您的家目录的绝对路径），然后将 `~/.local/bin` 添加到 `PATH` 中。您可以通过 [修改 `~/.profile`](#) 永久地设置 `PATH`。

在 Windows 上，您通过运行 `py -m site --user-site` 找到用户基础目录，然后将 `site-packages` 替换为 `Scripts`。比如，上述命令可能返回为 `C:\Users\Username\AppData\Roaming\Python36\site-packages`，然后您需要在 `PATH` 中包含 `C:\Users\Username\AppData\Roaming\Python36\Scripts`。您可以在 [控制面板 .aspx](#) 中永久设置用户的 `PATH`。您可能需要登出 `PATH` 更改才能生效。

为您的项目安装包

Pipenv 管理每个项目的依赖关系。要安装软件包时，请更改到您的项目目录（或只是本教程中的一个空目录）并运行：

```
1. $ cd myproject
2. $ pipenv install requests
```

Pipenv 将在您的项目目录中安装超赞的 `Requests` 库并为您创建一个 `Pipfile`。`Pipfile` 用于跟踪您的项目中需要重新安装的依赖，例如在与他人共享项目时。您应该得到类似的输出（尽管显示的确切路径会有所不同）：

```
1. Creating a Pipfile for this project...
2. Creating a virtualenv for this project...
3. Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/Versions/3.6'
4. New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.6
5. Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python
6. Installing setuptools, pip, wheel...done.
7.
8. Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
9. Installing requests...
10. Collecting requests
11. Using cached requests-2.18.4-py2.py3-none-any.whl
12. Collecting idna<2.7,>=2.5 (from requests)
13. Using cached idna-2.6-py2.py3-none-any.whl
14. Collecting urllib3<1.23,>=1.21.1 (from requests)
15. Using cached urllib3-1.22-py2.py3-none-any.whl
16. Collecting chardet<3.1.0,>=3.0.2 (from requests)
17. Using cached chardet-3.0.4-py2.py3-none-any.whl
18. Collecting certifi>=2017.4.17 (from requests)
19. Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
20. Installing collected packages: idna, urllib3, chardet, certifi, requests
21. Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4 urllib3-1.22
22.
23. Adding requests to Pipfile's [packages]...
24. P.S. You have excellent taste! ?
```

使用安装好的包

现在安装了 `Requests`，您可以创建一个简单的 `main.py` 文件来使用它：

```
1. import requests
2.
3. response = requests.get('https://httpbin.org/ip')
4.
5. print('Your IP is {}'.format(response.json()['origin']))
```

然后您就可以使用 `pipenv run` 运行这段脚本：

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

您应该获取到类似的输出：

```
1. Your IP is 8.8.8.8
```

使用 `$ pipenv run` 可确保您的安装包可用于您的脚本。我们还可以生成一个新的 shell，确保所有命令都可以使用 `$ pipenv shell` 访问已安装的包。

下一步

恭喜，您现在知道如何安装和使用Python包了！？

更低层次：virtualenv

`virtualenv` 是一个创建隔绝的Python环境的工具。`virtualenv` 创建一个包含所有必要的可执行文件的文件夹，用来使用Python工程所需的包。

它可以独立使用，代替Pipenv。

通过pip安装virtualenv：

```
1. $ pip install virtualenv
```

测试您的安装

```
1. $ virtualenv --version
```

基本使用

- 为一个工程创建一个虚拟环境：

```
1. $ cd my_project_folder
2. $ virtualenv my_project
```

`virtualenv my_project` 将会在当前的目录中创建一个文件夹，包含了Python可执行文件，以及 `pip` 库的一份拷贝，这样就能安装其他包了。虚拟环境的名字（此例中是 `my_project`）可以是任意的；若省略名字将会把文件均放在当前目录。

在任何您运行命令的目录中，这会创建Python的拷贝，并将之放在叫做 `my_project` 的文件中。

您可以选择使用一个Python解释器（比如 `python2.7`）：

```
1. $ virtualenv -p /usr/bin/python2.7 my_project
```

或者使用 `~/.bashrc` 的一个环境变量将解释器改为全局性的：

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

- 要开始使用虚拟环境，其需要被激活：

```
1. $ source my_project/bin/activate
```

当前虚拟环境的名字会显示在提示符左侧（比如说 `(my_project)您的电脑:您的工程 用户名$)`

以让您知道它是激活的。从现在起，任何您使用pip安装的包将会放在 ```my_project``` 文件夹中，与全局安装的Python隔绝开。

像平常一样安装包，比如：

```
1. $ pip install requests
```

- 如果您在虚拟环境中暂时完成了工作，则可以停用它：

```
1. $ deactivate
```

这将会回到系统默认的Python解释器，包括已安装的库也会回到默认的。

要删除一个虚拟环境，只需删除它的文件夹。（要这么做请执行 `rm -rf my_project`）

然后一段时间后，您可能会有很多个虚拟环境散落在系统各处，您将有可能忘记它们的名字或者位置。

其他注意

运行带 `-no-site-packages` 选项的 `virtualenv` 将不会包括全局安装的包。这可用于保持包列表干净，以防以后需要访问它。（这在 `virtualenv` 1.7及之后是默认行为）

为了保持您的环境的一致性，“冷冻住（freeze）”环境包当前的状态是个好主意。要这么做，请运行：

```
1. $ pip freeze > requirements.txt
```

这将会创建一个 `requirements.txt` 文件，其中包含了当前环境中所有包及各自的版本的简单列表。您可以使用“`pip list`”在不产生`requirements`文件的情况下，查看已安装包的列表。这将会使另一个不同的开发者（或者是您，如果您需要重新创建这样的环境）在以后安装相同版本的相同包变得容易。

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

这能帮助确保安装、部署和开发者之间的一致性。

最后，记住在源码版本控制中排除掉虚拟环境文件夹，可在`ignore`的列表中加上它。（查看 [版本控制忽略](#)）

virtualenvwrapper

[virtualenvwrapper](#)提供了一系列命令使得和虚拟环境工作变得愉快许多。它把您所有的虚拟环境都放在一个地方。

安装（确保 `virtualenv` 已经安装了）：

```
1. $ pip install virtualenvwrapper
2. $ export WORKON_HOME=~/Envs
3. $ source /usr/local/bin/virtualenvwrapper.sh
```

(`virtualenvwrapper` 的完整安装指引。)

对于Windows，您可以使用 `virtualenvwrapper-win`。

安装（确保 `virtualenv` 已经安装了）：

```
1. $ pip install virtualenvwrapper-win
```

在Windows中，`WORKON_HOME`默认的路径是 `%USERPROFILE%Envs`。

基本使用

- 创建一个虚拟环境：

```
1. $ mkvirtualenv my_project
```

这会在 `~/Envs` 中创建 `my_project` 文件夹。

- 在虚拟环境上工作：

```
1. $ workon my_project
```

或者，您可以创建一个项目，它会创建虚拟环境，并在 `$WORKON_HOME` 中创建一个项目目录。当您使用 `workon myproject` 时，会 `cd` 到项目目录中。

```
1. $ mkproject myproject
```

`virtualenvwrapper` 提供环境名字的tab补全功能。当您有很多环境，并且很难记住它们的名字时，这就显得很有用。

`workon` 也能停止您当前所在的环境，所以您可以在环境之间快速的切换。

- 停止是一样的：

```
1. $ deactivate
```

- 删除：

```
1. $ rmvirtualenv my_project
```

其他有用的命令

lsvirtualenv

列举所有的环境。

cdvirtualenv

导航到当前激活的虚拟环境的目录中，比如说这样您就能够浏览它的 **site-packages** 。

cdsitepackages

和上面的类似，但是是直接进入到 **site-packages** 目录中。

lssitepackages

显示 **site-packages** 目录中的内容。

[virtualenvwrapper 命令的完全列表](#) 。

virtualenv-burrito

有了 **virtualenv-burrito**，您就能使用单行命令拥有virtualenv + virtualenvwrapper的环境。

autoenv

当您 **cd** 进入一个包含 **.env** 的目录中，就会 **autoenv**自动激活那个环境。

使用 **brew** 在Mac OS X上安装它：

```
1. $ brew install autoenv
```

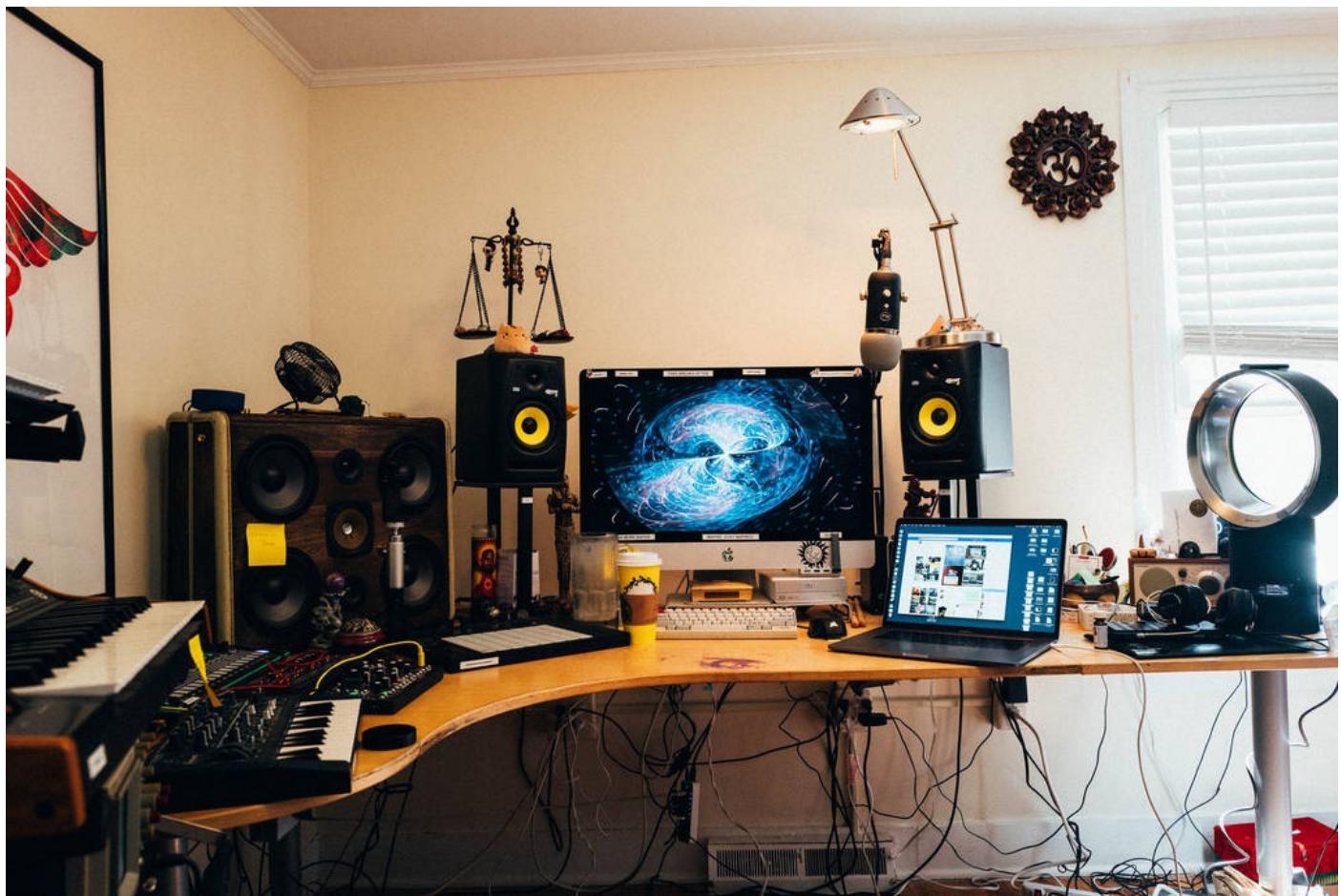
在Linux上：

```
1. $ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
2. $ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

原文：<http://pythonguidecn.readthedocs.io/zh/latest/dev/virtualenvs.html>

- 您的开发环境
- Pipenv & 虚拟环境
- Pip和Virtualenv的更多配置

您的开发环境



文本编辑器

任何能够编辑普通文本的编辑器都能够用来编写Python代码，然后，使用一个更加强大的编辑器可能使您的生活变得容易点。

Vim

Vim是一个使用键盘快捷键而不是菜单或图标来编辑的文本编辑器。有许多增强Vim编辑器中Python开发环境的插件和设置。如果您只开发Python，使用缩进和换行均符合 [PEP 8](#)要求的默认设置是一个好的开始。在您的home目录中，打开 `.vimrc` 文件，添加下面这些内容：

```
1. set textwidth=79 " lines longer than 79 columns will be broken
2. set shiftwidth=4 " operation >> indents 4 columns; << unindents 4 columns
3. set tabstop=4      " a hard TAB displays as 4 columns
4. set expandtab      " insert spaces when hitting TABs
5. set softtabstop=4 " insert/delete 4 spaces when hitting a TAB/BACKSPACE
6. set shiftround     " round indent to multiple of 'shiftwidth'
7. set autoindent     " align the new line indent with the previous line
```

基于上述设置，新行会在超过79个字符被添加，`tab`键则会自动转换为4个空格。如果您还使用Vim编辑其他语言，有一个叫做 `indent` 的便捷插件可以让这个设置只为Python源文件服务。

还有一个方便的语法插件叫做 `syntax`，改进了Vim 6.1中的语法文件。

这些插件使您拥有一个基本的环境进行Python开发。要最有效的使用Vim，您应该时常检查代码的语法错误和是否符合PEP8。幸运的是，`pycodestyle` 和 `Pyflakes` 将会帮您做这些。如果您的Vim是用 `+python` 编译的，您也可以在编辑器中使用一些非常有用的插件来做这些检查。

对于PEP8检查和pyflakes，您可以安装 `vim-flake8`。然后您就可以在Vim中把 `Flake8` 映射到任何热键或您想要的行为上。这个插件将会在屏幕下方显示出错误，并且提供一个简单的方式跳转到相关行。在保存文件的时候调用这个功能会是非常方便的。要这么做，就把下面一行加入到您的 `.vimrc`：

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

如果您已经在使用 `syntastic`，您可以设置它来运行Pyflakes，并在quickfix窗口中显示错误和警告。一个这样做并还会在状态栏中显示状态和警告信息的样例是：

```
1. set statusline+=%#warningmsg#
2. set statusline+=%{SyntasticStatuslineFlag()}
3. set statusline+=%*
4. let g:syntastic_auto_loc_list=1
5. let g:syntastic_loc_list_height=5
```

Python-mode

`Python-mode` 是一个在Vim中使用Python的综合解决方案。它拥有：

- 任意组合的异步Python代码检查（`pylint`、`pyflakes`、`pycodestyle`、`mccabe`）
- 使用Rope进行代码重构和补全
- Python快速折叠
- 支持`virtualenv`
- 搜索Python文档，运行Python代码
- 自动修复 `pycodestyle` 错误
- 以及其他更多。

SuperTab

`SuperTab` 是一个小的Vim插件，通过使用 `<Tab>` 或任何其他定制的按键，能够使代码补全变得更方便。

Emacs

Emacs是另一个强大的文本编辑器。它是完全可编程的（lisp），但要正确的工作要花些功夫。如果您已经是一名Emacs的用户了，在EmacsWiki上的 [Python Programming in Emacs](#)将会是好的开始。

- Emacs 本身支持Python模式。

TextMate

[TextMate](#) 将苹果操作系统技术带入了文本编辑器的世界。通过桥接UNIX和GUI，TextMate将两者中最好的部分带给了脚本专家和新手用户。

Sublime Text

[Sublime Text](#) 是一款高级的，用来编写代码、标记和文章的文本编辑器。您将会爱上漂亮的用户界面、非凡的特性和惊人的表现。

Sublime Text对编写Python代码支持极佳，而且它使用Python写其插件API。它也拥有大量各式各样的插件，[其中一些](#)允许编辑器内的PEP8检查和代码提示。

Atom

[Atom](#) 是一款21世纪的可删减的(hackable)文本编辑器。它基于我们所喜欢的编辑器的任何优秀特性，并构建于atom-shell上。

Atom是web原生的(HTML、CSS、JS)，专注于模块化的设计和简单的插件开发。它自带本地包管理和大量的包。Python开发所推荐的插件是[Linter](#)和[linter-flake8](#)的组合。

IDEs

PyCharm / IntelliJ IDEA

[PyCharm](#) 由JetBrains公司开发，此公司还以IntelliJ IDEA闻名。它们都共享着相同的基础代码，PyCharm中大多数特性能通过免费的[Python 插件](#)带入到IntelliJ中。PyCharm由两个版本：专业版(Professional Edition)(30天试用)和拥有相对少特性的社区版(Community Edition)(Apache 2.0 License)。

Python (在 Visual Studio Code 中)

[用于Visual Studio的Python](#)是一款用于[Visual Studio Code IDE](#)的扩展。它是一个免费的、轻量的、开源的IDE，支持Mac、Windows和Linux。它以诸如Node.js和Python等开源技术构建，具有如Intellisense(自动补全)、本地和远程调试、linting(代码检查)等引人注目的特性。

MIT 许可证。

Enthought Canopy

[Enthought Canopy](#) 是一款专门面向科学家和工程师的Python IDE，它预装了为数据分析而用的库。

Eclipse

Eclipse中进行Python开发最流行的插件是Aptana的[PyDev](#)。

Komodo IDE

[Komodo IDE](#) 由ActiveState开发，并且是在Windows、Mac和Linux平台上的商业IDE。

Spyder

[Spyder](#) 是一款专门面向和Python科学库（即 [Scipy](#) ）打交道的IDE。它集成了 [pyflakes](#)、[pylint](#) 和 [rope](#)。

Spyder是开源的（免费的），提供了代码补全、语法高亮、类和函数浏览器，以及对象检查的功能。

WingIDE

[WingIDE](#) 是一个专门面向Python的IDE。它能运行在Linux、Windows和Mac（作为一款X11应用程序，会使某些Mac用户遇到困难）上。

WingIDE提供了代码补全、语法高亮、源代码浏览器、图形化调试器的功能，还支持版本控制系统。

NINJA-IDE

[NINJA-IDE](#) （来自递归缩写：“Ninja-IDE Is Not Just Another IDE”）是一款跨平台的IDE，特别设计成构建Python应用，并能运行于Linux/X11、Mac OS X和Windows桌面操作系统上。从网上可以下载到这些平台的安装包。

NINJA-IDE是一款开源软件（GPLv3许可），是使用Python和Qt开发。在 [GitHub](#)能下载到源文件。

Eric (The Eric Python IDE)

[Eric](#) 是一款功能齐全的Python IDE，提供源代码自动补全、语法高亮、对版本控制系统的支持、对Python 3的支持、集成的web浏览器、Python Shell、集成的调试器和灵活的插件系统等功能。它基于Qt GUI工具集，使用Python编写，集成了Scintilla编辑器控制。Eric是一款超过10年活跃开发的开源软件工程（GPLv3许可）。

解释器工具

虚拟环境

虚拟环境提供了隔离项目包依赖的强大方式。这意味着您无须再系统范围内安装Python工程特定的包，因此就能避免潜在的版本冲突。

To start using and see more information: [Virtual Environments](#) docs. 开始使用和查阅更多信息：
请参阅 [Virtual Environments](#) 文档。

pyenv

[pyenv](#) 是一个允许多个Python解释器版本同时安装于一台机器的工具。这解决了不同的项目需要不同版本的Python的问题。比如，为了兼容性，可以很容易地为一个项目安装Python 2.7，而继续使用Python 3.4作为默认的编辑

器。pyenv不止限于CPython版本—它还能安装PyPy、anaconda、miniconda、stackless、jython和ironpython解释器。

pyenv的工作原理是在一个叫做 `shims` 目录中创建Python解释器（以及其他工具像 `pip` 和 `2to3` 等）的假版本。当系统寻找名为 `python` 的应用时，它会先在 `shims` 目录中查找，并使用那个假版本，然后会传递命令到pyenv中。pyenv基于环境变量、`.python-version` 文件和全局默认设置的信息就知道该运行哪个版本的Python。

pyenv不是管理虚拟环境的工具，但是有一个叫做 `pyenv-virtualenv` 的插件可以自动化不同环境的创建，而且也能够使用现有的pyenv工具，基于环境变量或者 `.python-version` 文件，来切换不同的环境。

其他工具

IDLE

`IDLE` 是一个集成的开发环境，它是Python标准库的一部分。它完全由Python编写，并使用Tkinter GUI工具集。尽管IDLE不适用于作为成熟的Python开发工具，但它对尝试小的Python代码和对Python不同特性的实验非常有帮助。

它提供以下特性：

- Python Shell窗口（解释器）
- 多窗口文本编辑器，支持彩色化Python代码
- 最小的调试工具

IPython

`IPython` 提供一个丰富的工具集来帮助您最大限度地和Python交互。它主要的组件有：

- 强大的Python shell（终端和基于Qt）。
- 一个基于网络的笔记本，拥有相同的核心特性，但是支持富媒体、文本、代码、数学表达式和内联绘图。
- 支持交互式的数据可视化和GUI工具集的使用。
- 灵活、嵌入的解释器载入到您的工程工程中。
- 支持高级可交互的并行计算的工具。

```
1. $ pip install ipython
```

下载和安装带有所有可选依赖（notebook、qtconsole、tests和其他功能）的IPython

```
1. $ pip install ipython[all]
```

BPython

`bpython` 在类Unix操作系统中可替代Python解释器的接口。它有以下特性：

- 内联的语法高亮。

- 行内输入时的自动补全建议。
- 任何Python函数的期望参数列表。
- 从内存中pop出代码的最后一行并重新运行 (re-evaluate) 的“倒带”功能。
- 将输入的代码发送到pastebin。
- 将输入的代码保存到一个文件中。
- 自动缩进。
- 支持Python 3。

```
1. $ pip install bpython
```

ptpython

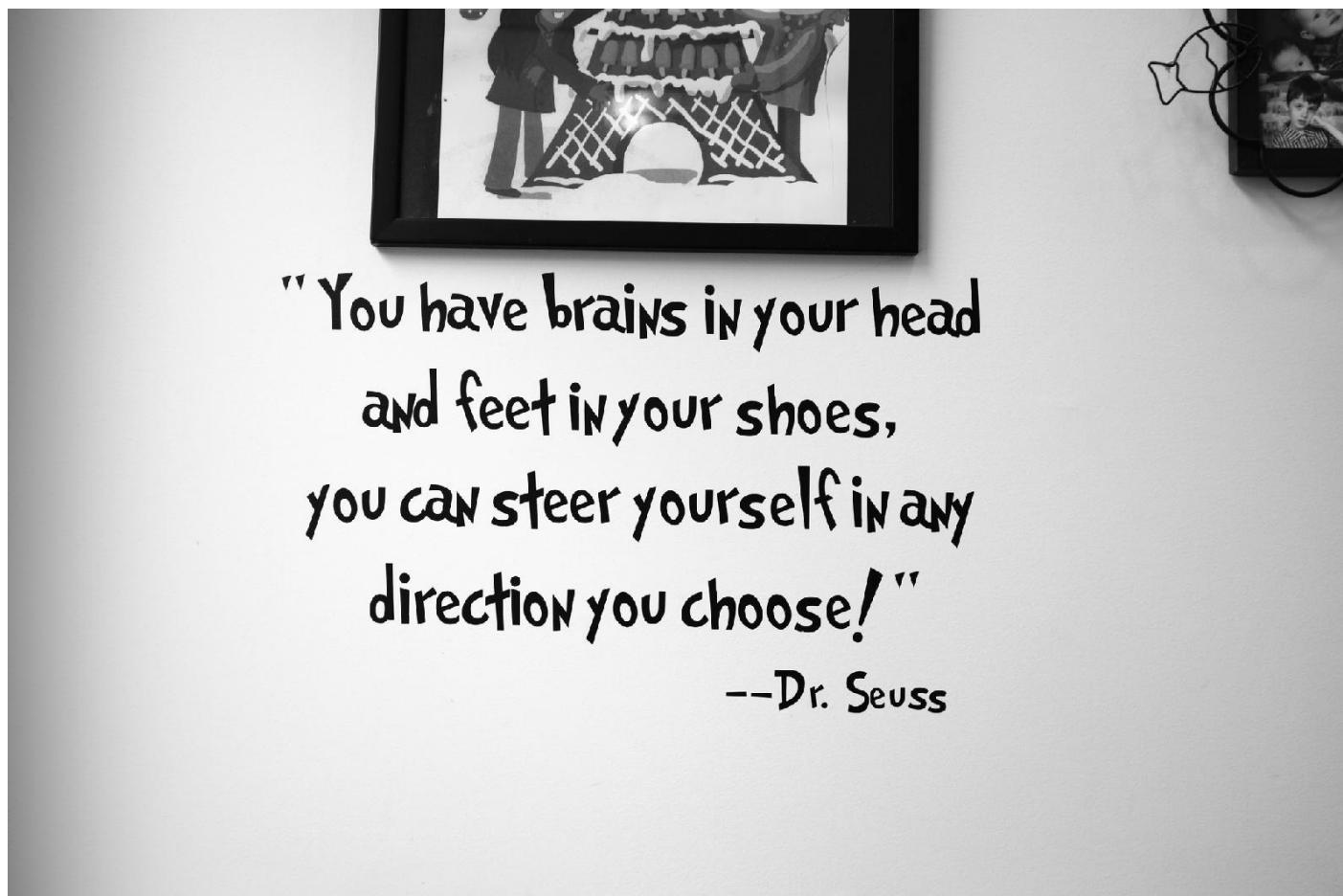
`ptpython` 是一个构建在[prompt_toolkit](#)库顶部的REPL。它被视作是 [BPython](#) 的替代。特性包括：

- 语法高亮
- 自动补全
- 多行编辑
- Emacs和VIM模式
- 代码中嵌入的REPL
- 语法合法性
- Tab页
- 通过安装Ipython `pip install ipython` 并运行 `ptipython`，支持集成 [IPython](#) 的 shell

```
1. $ pip install ptpython
```

原文：<http://pythonguidecn.readthedocs.io/zh/latest/dev/env.html>

Pipenv & 虚拟环境



本教程将引导您完成安装和使用 Python 包。

它将向您展示如何安装和使用必要的工具，并就最佳做法做出强烈推荐。请记住，Python 用于许多不同的目的。准确地说，您希望如何管理依赖项可能会根据您如何决定发布软件而发生变化。这里提供的指导最直接适用于网络服务（包括 Web 应用程序）的开发和部署，但也非常适合管理任意项目的开发和测试环境。

注解

本指南是针对 Python 3 编写。但如果由于某种原因仍然使用 Python 2.7，这些指引应该能够正常工作。

确保您已经有了 Python 和 pip

在您进一步之前，请确保您有 Python，并且可从您的命令行中获得。您可以通过简单地运行以下命令来检查：

```
1. $ python --version
```

您应该得到像 [3.6.2](#) 之类的一些输出。如果没有 Python，请从 python.org 安装最新的 3.x 版本，或参考本指南的 [安装 Python](#) 一节。

注解

如果您是新手，您会得到如下错误：

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

这是因为此命令要在 `shell`（也称为 终端 或 控制台）中运行。有关使用操作系统的`shell` 并和 Python 进行交互的介绍，请参阅面向 Python 新手的 [入门教程](#)。

另外，您需要确保 `pip` 是可用的。您可以通过运行以下命令来检查：

```
1. $ pip --version
```

如果您使用 [python.org](#) 或 [Homebrew](#) 的安装程序来安装 Python，您应该已经有 `pip` 了。如果您使用的是 Linux，并使用操作系统的包管理器进行安装，则可能需要单独[安装 pip](#)。

安装 Pipenv

[Pipenv](#) 是 Python 项目的依赖管理器。如果您熟悉 Node.js 的 `npm` 或 Ruby 的 `bundler`，那么它们在思路上与这些工具类似。尽管 `pip` 可以安装 Python 包，但仍推荐使用 Pipenv，因为它是一种更高级的工具，可简化依赖关系管理的常见使用情况。

使用 `pip` 来安装 Pipenv：

```
1. $ pip install --user pipenv
```

注解

这进行了 [用户安装](#)，以防止破坏任何系统范围的包。如果安装后，`shell` 中没有 `pipenv`，则需要将 [用户基础目录](#) 的二进制文件目录添加到 `PATH` 中。

在 Linux 和 macOS 上，您可以通过运行 `python -m site --user-base` 找到用户基础目录，然后把 `bin` 加到目录末尾。比如，上述命令典型地会打印出 `~/.local/bin`（`~` 会扩展为您的家目录的绝对路径），然后将 `~/.local/bin` 添加到 `PATH` 中。您可以通过 [修改 `~/.profile`](#) 永久地设置 `PATH`。

在 Windows 上，您通过运行 `py -m site --user-site` 找到用户基础目录，然后将 `site-packages` 替换为 `Scripts`。比如，上述命令可能返回为 `C:\Users\Username\AppData\Roaming\Python36\site-packages`，然后您需要在 `PATH` 中包含 `C:\Users\Username\AppData\Roaming\Python36\Scripts`。您可以在 [控制面板 .aspx](#) 中永久设置用户的 `PATH`。您可能需要登出 `PATH` 更改才能生效。

为您的项目安装包

Pipenv 管理每个项目的依赖关系。要安装软件包时，请更改到您的项目目录（或只是本教程中的一个空目录）并运行：

```
1. $ cd myproject
2. $ pipenv install requests
```

Pipenv 将在您的项目目录中安装超赞的 `Requests` 库并为您创建一个 `Pipfile`。`Pipfile` 用于跟踪您的项目中需要重新安装的依赖，例如在与他人共享项目时。您应该得到类似的输出（尽管显示的确切路径会有所不同）：

```
1. Creating a Pipfile for this project...
2. Creating a virtualenv for this project...
3. Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/Versions/3.6'
4. New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.6
5. Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python
6. Installing setuptools, pip, wheel...done.
7.
8. Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
9. Installing requests...
10. Collecting requests
11.   Using cached requests-2.18.4-py2.py3-none-any.whl
12. Collecting idna<2.7,>=2.5 (from requests)
13.   Using cached idna-2.6-py2.py3-none-any.whl
14. Collecting urllib3<1.23,>=1.21.1 (from requests)
15.   Using cached urllib3-1.22-py2.py3-none-any.whl
16. Collecting chardet<3.1.0,>=3.0.2 (from requests)
17.   Using cached chardet-3.0.4-py2.py3-none-any.whl
18. Collecting certifi>=2017.4.17 (from requests)
19.   Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
20. Installing collected packages: idna, urllib3, chardet, certifi, requests
21. Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4 urllib3-1.22
22.
23. Adding requests to Pipfile's [packages]...
24. P.S. You have excellent taste! ?
```

使用安装好的包

现在安装了 `Requests`，您可以创建一个简单的 `main.py` 文件来使用它：

```
1. import requests
2.
3. response = requests.get('https://httpbin.org/ip')
4.
5. print('Your IP is {}'.format(response.json()['origin']))
```

然后您就可以使用 `pipenv run` 运行这段脚本：

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

您应该获取到类似的输出：

```
1. Your IP is 8.8.8.8
```

使用 `$ pipenv run` 可确保您的安装包可用于您的脚本。我们还可以生成一个新的 shell，确保所有命令都可以使用 `$ pipenv shell` 访问已安装的包。

下一步

恭喜，您现在知道如何安装和使用Python包了！？

更低层次：virtualenv

`virtualenv` 是一个创建隔绝的Python环境的工具。`virtualenv` 创建一个包含所有必要的可执行文件的文件夹，用来使用Python工程所需的包。

它可以独立使用，代替Pipenv。

通过pip安装virtualenv：

```
1. $ pip install virtualenv
```

测试您的安装

```
1. $ virtualenv --version
```

基本使用

- 为一个工程创建一个虚拟环境：

```
1. $ cd my_project_folder
2. $ virtualenv my_project
```

`virtualenv my_project` 将会在当前的目录中创建一个文件夹，包含了Python可执行文件，以及 `pip` 库的一份拷贝，这样就能安装其他包了。虚拟环境的名字（此例中是 `my_project`）可以是任意的；若省略名字将会把文件均放在当前目录。

在任何您运行命令的目录中，这会创建Python的拷贝，并将之放在叫做 `my_project` 的文件中。

您可以选择使用一个Python解释器（比如 `python2.7`）：

```
1. $ virtualenv -p /usr/bin/python2.7 my_project
```

或者使用 `~/.bashrc` 的一个环境变量将解释器改为全局性的：

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

- 要开始使用虚拟环境，其需要被激活：

```
1. $ source my_project/bin/activate
```

当前虚拟环境的名字会显示在提示符左侧（比如说 `(my_project)您的电脑:您的工程 用户名$`）

以让您知道它是激活的。从现在起，任何您使用pip安装的包将会放在 ```my_project``` 文件夹中，与全局安装的Python隔绝开。

像平常一样安装包，比如：

```
1. $ pip install requests
```

- 如果您在虚拟环境中暂时完成了工作，则可以停用它：

```
1. $ deactivate
```

这将会回到系统默认的Python解释器，包括已安装的库也会回到默认的。

要删除一个虚拟环境，只需删除它的文件夹。（要这么做请执行 `rm -rf my_project`）

然后一段时间后，您可能会有很多个虚拟环境散落在系统各处，您将有可能忘记它们的名字或者位置。

其他注意

运行带 `-no-site-packages` 选项的 `virtualenv` 将不会包括全局安装的包。这可用于保持包列表干净，以防以后需要访问它。（这在 `virtualenv` 1.7及之后是默认行为）

为了保持您的环境的一致性，“冷冻住（freeze）”环境包当前的状态是个好主意。要这么做，请运行：

```
1. $ pip freeze > requirements.txt
```

这将会创建一个 `requirements.txt` 文件，其中包含了当前环境中所有包及各自的版本的简单列表。您可以使用“`pip list`”在不产生`requirements`文件的情况下，查看已安装包的列表。这将会使另一个不同的开发者（或者是您，如果您需要重新创建这样的环境）在以后安装相同版本的相同包变得容易。

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

这能帮助确保安装、部署和开发者之间的一致性。

最后，记住在源码版本控制中排除掉虚拟环境文件夹，可在`ignore`的列表中加上它。（查看 [版本控制忽略](#)）

virtualenvwrapper

[virtualenvwrapper](#)提供了一系列命令使得和虚拟环境工作变得愉快许多。它把您所有的虚拟环境都放在一个地方。

安装（确保 `virtualenv` 已经安装了）：

```
1. $ pip install virtualenvwrapper
2. $ export WORKON_HOME=~/Envs
3. $ source /usr/local/bin/virtualenvwrapper.sh
```

(`virtualenvwrapper` 的完整安装指引。)

对于Windows，您可以使用 `virtualenvwrapper-win`。

安装（确保 `virtualenv` 已经安装了）：

```
1. $ pip install virtualenvwrapper-win
```

在Windows中，`WORKON_HOME`默认的路径是 `%USERPROFILE%Envs`。

基本使用

- 创建一个虚拟环境：

```
1. $ mkvirtualenv my_project
```

这会在 `~/Envs` 中创建 `my_project` 文件夹。

- 在虚拟环境上工作：

```
1. $ workon my_project
```

或者，您可以创建一个项目，它会创建虚拟环境，并在 `$WORKON_HOME` 中创建一个项目目录。当您使用 `workon myproject` 时，会 `cd` 到项目目录中。

```
1. $ mkproject myproject
```

`virtualenvwrapper` 提供环境名字的tab补全功能。当您有很多环境，并且很难记住它们的名字时，这就显得很有用。

`workon` 也能停止您当前所在的环境，所以您可以在环境之间快速的切换。

- 停止是一样的：

```
1. $ deactivate
```

- 删除：

```
1. $ rmvirtualenv my_project
```

其他有用的命令

lsvirtualenv

列举所有的环境。

cdvirtualenv

导航到当前激活的虚拟环境的目录中，比如说这样您就能够浏览它的 **site-packages** 。

cdsitepackages

和上面的类似，但是是直接进入到 **site-packages** 目录中。

lssitepackages

显示 **site-packages** 目录中的内容。

[virtualenvwrapper 命令的完全列表](#) 。

virtualenv-burrito

有了 **virtualenv-burrito**，您就能使用单行命令拥有virtualenv + virtualenvwrapper的环境。

autoenv

当您 **cd** 进入一个包含 **.env** 的目录中，就会 **autoenv**自动激活那个环境。

使用 **brew** 在Mac OS X上安装它：

```
1. $ brew install autoenv
```

在Linux上：

```
1. $ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
2. $ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

原文：<http://pythonguidecn.readthedocs.io/zh/latest/dev/virtualenvs.html>

Pip和Virtualenv的更多配置



用 pip 来要求一个激活的虚拟环境

现在应该很清楚了，使用虚拟环境是个保持开发环境干净和分隔不同项目要求的好做法。

当您开始工作在多个不同的项目上时，会很难记住去激活哪个相关的虚拟环境来回到特定的项目。其结果就是，您会非常容易在全局范围内安装包，虽然想的是要安装在特定工程的虚拟环境中。时间越久，就会导致混乱的全局包列表。

为了确保您当您使用 `pip install` 时是将包安装在激活的虚拟环境中，考虑在 `~/.bashrc` 文件中加上以下一行：

```
1. export PIP_REQUIRE_VIRTUALENV=true
```

在保存完这个修改以及使用 `source ~/.bashrc` 来source一下 `~/.bashrc` 文件后，如果您不在一个虚拟环境中，`pip`就不会让您安装包。如果您试着在虚拟环境外使用 `pip install`，`pip`将会柔地提示您需要一个激活的虚拟环境来安装包。

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

您也可以通过编辑 `pip.conf` 或 `pip.ini` 来做相同的配置。  `pip.conf` 被Unix和Mac OS X操作系统使用，能够在这里找到：

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

类似的，`pip.ini` 被Windows操作系统使用，能够在这里找到：

```
1. %HOMEP\pip\pip.ini
```

如果在这些位置中并没有 `pip.conf` 或 `pip.ini`，您可以在对应的操作系统中创建一个正确名字的新文件。

如果您早就拥有配置文件了，只需将下行添加到 `[global]` 设置下，即可要求一个激活的虚拟环境：

```
1. require-virtualenv = true
```

如果您没有配置文件，您需要创建一个新的，然后把下面几行添加到这个新文件中：

```
1. [global]
2. require-virtualenv = true
```

当然，您也需要在全局范围内安装一些包（通常是在多个项目中都要一直用到的包），可以添加下面内容到

`~/.bashrc` 来完成：

```
1. gpip() {
2.     PIP_REQUIRE_VIRTUALENV="" pip "$@"
3. }
```

在保存完这个修改以及使用 `source ~/.bashrc` 来source一下 `~/.bashrc` 文件后，您现在可以通过运行 `gpip install` 来在全局范围内安装包。您可以把函数名改成任何您喜欢的，只要记住当您要用pip在全局范围内安装包的时候使用那个名字就行了。

存下包以供将来使用

每个开发者都有偏好的库，当您工作在大量不同的项目上时，这些项目之间肯定有一些重叠的库。比如说，您可能在多个不同的项目上使用了 `requests`。

每当您开始一个新项目（并有一个新的虚拟环境）重新下载相同的包/库是没有必要的。幸运的是，自从6.0版本开始，pip提供 `默认缓存机制` 而无需任何配置。

当使用更老的版本时，你可以用下面的方式来配置pip，以使它尝试重用已安装的包。

在UNIX系统中，您可以添加以下两行到您的 `.bashrc` 或 `.bash_profile` 文件中。

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

您可以设置成任何您喜欢的路径（只要设置了写权限）。添加完后，`source` 下您的`.bashrc`（或者`.bash_profile`）文件，就设置好啦。

另一个进行相同配置的方法是通过`pip.conf` 或`pip.ini` 文件来做，这取决于您的系统。如果您用Windows，就将下面一行添加到`pip.ini` 文件中的`[global]` 设置下：

```
1. download-cache = %HOME%\pip\cache
```

类似的，如果您使用UNIX，就将下面一行添加到`pip.conf` 文件中的`[global]` 设置下：

```
1. download-cache = $HOME/.pip/cache
```

虽然您可以使用任何您喜欢的存储缓存的路径，但是仍然推荐在`pip.conf` 或者`pip.ini` 文件所在目录下床架一个新的文件夹`in`。如果您不相信自己能够处理好这个路径，就使用这里提供的内容就好，不会有问题是的。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/dev/pip-virtualenv.html>

- 结构化您的工程
- 代码风格
- 阅读好的代码
- 文档
- 测试您的代码
- 日志 (Logging)
- 常见陷阱
- 选择一个许可

结构化您的工程



我们对于“结构化”的定义是您关注于怎样使您的项目最好地满足它的对象性，我们需要去考虑如何更好地利用Python的特性来创造简洁、高效的代码。在实践层面，“结构化”意味着通过编写简洁的代码，并且正如文件系统中文件和目录的组织一样，代码应该使逻辑和依赖清晰。

哪个函数应该深入到哪个模块？数据在项目中如何流转？什么功能和函数应该组合或独立？要解决这些问题，您可以开始做个一计划，大体来说，即是您的最终产品看起来会是怎样的。

在这一章节中，我们更深入地去观察Python的模块和导入系统，因为它们是加强您的项目结构化的关键因素，接着我们会从不同层面去讨论如何去构建可扩展且测试可靠的代码。

仓库的结构

这很重要

在一个健康的开发周期中，代码风格，API设计和自动化是非常关键的。同样的，对于工程的 架构，仓库的结构也是关键的一部分。

当一个潜在的用户和贡献者登录到您的仓库页面时，他们会看到这些：

- 工程的名字
- 工程的描述

- 一系列的文件

只有当他们滚动到目录下方时才会看到您工程的README。

如果您的仓库的目录是一团糟，没有清晰的结构，他们可能要到处寻找才能找到您写的漂亮的文档。

为您的渴望的事业而奋斗，而不是仅仅只为您现在的工作而工作。

当然，第一印象并不是一切。但是，您和您的同事会和这个仓库并肩战斗很长时间，会熟悉它的每一个角落和细节。拥有良好的布局，事半功倍。

仓库样例

请看这里：这是 [Kenneth Reitz](#) 推荐的。

这个仓库 [可以在Github上找到](#) 。

```
1. README.rst
2. LICENSE
3. setup.py
4. requirements.txt
5. sample/__init__.py
6. sample/core.py
7. sample/helpers.py
8. docs/conf.py
9. docs/index.rst
10. tests/test_basic.py
11. tests/test_advanced.py
```

让我们看一下细节。

真正的模块

| 布局 | [./sample/](#) or [./sample.py](#)
| 作用 | 核心代码

您的模块包是这个仓库的核心，它不应该隐藏起来：

```
1. ./sample/
```

如果您的模块只有一个文件，那么您可以直接将这个文件放在仓库的根目录下：

```
1. ./sample.py
```

这个模块文件不应该属于任何一个模棱两可的src或者python子目录。

License

| 布局 | [./LICENSE](#)

|作用|许可证.

除了源代码本身以外，这个毫无疑问是您仓库最重要的一部分。在这个文件中要有完整的许可说明和授权。

如果您不太清楚您应该使用哪种许可方式，请查看 [choosealicense.com](#).

当然，您也可以在发布您的代码时不做任何许可说明，但是这显然阻碍潜在的用户使用您的代码。

Setup.py

|布局| `./setup.py`

|作用|打包和发布管理

如果您的模块包在您的根目录下，显然这个文件也应该在根目录下。

Requirements File

|布局| `./requirements.txt`

|作用|开发依赖.

一个 `pip requirements file` 应该放在仓库的根目录。它应该指明完整工程的所有依赖包：测试，编译和文档生成。

如果您的工程没有任何开发依赖，或者您喜欢通过 `setup.py` 来设置，那么这个文件不是必须的。

Documentation

|布局| `./docs/`

|作用|包的参考文档

没有任何理由把这个放到别的地方。

Test Suite

想了解关于编写测试的建议，请查阅 [/writing/tests](#)。

|布局| `./test_sample.py` or `./tests`

|作用|包的集合和单元测试

最开始，一组测试例子只是放在一个文件当中：

```
1. ./test_sample.py
```

当测试例子逐步增加时，您会把它放到一个目录里面，像下面这样：

```
1. tests/test_basic.py  
2. tests/test_advanced.py
```

当然，这些测试例子需要导入您的包来进行测试，有几种方式来处理：

- 将您的包安装到site-packages中。
- 通过简单直接的路径设置来解决导入的问题。

我极力推荐后者。如果使用 `setup.py develop` 来测试一个持续更新的代码库，需要为每一个版本的代码库设置一个独立的测试环境。太麻烦了。

可以先创建一个包含上下文环境的文件 `tests/context.py` file:

```
1. import os
2. import sys
3. sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
4.
5. import sample
```

然后，在每一个测试文件中，导入：

```
1. from .context import sample
```

这样就能够像期待的那样工作，而不用采用安装的方式。

一些人会说应该把您的测试例子放到您的模块里面 – 我不同意。这样会增加您用户使用的复杂度；而且添加测试模块将导致需要额外的依赖和运行环境。

Makefile

| 布局 | `./Makefile`

| 作用 | 常规的管理任务

如果您看看我的项目或者其他开源项目，您都会发现有一个Makefile。为什么？这些项目也不是用C写的啊。。。简而言之，make对于定义常规的管理任务是非常有用的工具。

样例 Makefile：

```
1. init:
2.     pip install -r requirements.txt
3.
4. test:
5.     py.test tests
6.
7. PHONY: init test
```

一些其他的常规管理脚本（比如 `manage.py` 或者 `fabfile.py`），也放在仓库的根目录下。

关于 Django Applications

从Django 1.4开始，我发现有这样一个现象：很多开发者错误地使用Django自带的应用模板创建项目，导致他们的仓库结构非常糟糕。

这是怎么回事儿？是的，他们在进入一个新的仓库后，通常都这样操作：

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

这样的操作生成的仓库结构是这样的：

```
1. README.rst
2. samplesite/manage.py
3. samplesite/samplesite/settings.py
4. samplesite/samplesite/wsgi.py
5. samplesite/samplesite/sampleapp/models.py
```

亲，不要这样做。

相对路径会让您的工具和您的开发者都很疑惑。没有必要的嵌套对任何人都没有好处（除非您怀念庞大的SVN仓库）。

让我们这样来做：

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

注意末尾的 ". "。

生成的结构是这样的：

```
1. README.rst
2. manage.py
3. samplesite/settings.py
4. samplesite/wsgi.py
5. samplesite/sampleapp/models.py
```

结构是一把钥匙

得益于Python提供的导入与管理模块的方式，结构化Python项目变得相对简单。这里说的简单，指的是结构化过程没有太多约束限制而且模块导入功能容易掌握。因而您只剩下架构性的工作，包括设计、实现项目各个模块，并整理清他们之间的交互关系。

容易结构化的项目同样意味着它的结构化容易做得糟糕。糟糕结构的特征包括：

- **多重且混乱的循环依赖关系：**假如在 `furn.py` 内的Table与Chair类需要导入 `workers.py` 中的Carpenter类以回答类似 `table.isdoneby()` 的问题，并且Carpenter类需要引入Table和Chair类以回答 `carpenter.whatdo()` 这类问题，这就是一种循环依赖的情况。在这种情况下，您得借助一些不怎么靠谱的小技巧，比如在方法或函数内部使用import语句。
- **隐含耦合：**Table类实现代码中每一个改变都会打破20个不相关的测试用例，由于它影响了Carpenter类的代码，这要求谨慎地操作以适应改变。这样的情况意味着Carpenter类代码中包含了太多关于Table类的假设关联（或相反）。
- **大量使用全局变量或上下文：**如果Table和Carpenter类使用不仅能被修改而且能被不同引用修改的全局变量，而不是明确地传递 (`height, width, type, wood`) 变量。您就需要彻底检查全局变量的所有入口，来

理解到为什么一个长方形桌子变成了正方形，最后发现远程的模板代码修改了这份上下文，弄错了桌子尺寸规格的定义。

- 面条式代码 (Spaghetti code)：多页嵌套的if语句与for循环，包含大量复制-粘贴的过程代码，且没有合适的分割—这样的代码被称为面条式代码。Python中有意思的缩进排版(最具争议的特性之一)使面条式代码很难维持。所以好消息是您也许不会经常看到这种面条式代码。
- Python中更可能出现混沌代码：这类代码包含上百段相似的逻辑碎片，通常是缺乏合适结构的类或对象，如果您始终弄不清手头上的任务应该使用FurnitureTable, AssetTable还是Table，甚至TableNew，也许您已经陷入了混沌代码中。

模块

Python模块是最主要的抽象层之一，并且很可能是最自然的一个。抽象层允许将代码分为不同部分，每个部分包含相关的数据与功能。

例如在项目中，一层控制用户操作相关接口，另一层处理底层数据操作。最自然分开这两层的方式是，在一份文件里重组所有功能接口，并将所有底层操作封装到另一个文件中。这种情况下，接口文件需要导入封装底层操作的文件，可通过 `import` 和 `from ... import` 语句完成。一旦您使用 `import` 语句，就可以使用这个模块。既可以是内置的模块包括 `os` 和 `sys`，也可以是已经安装的第三方的模块，或者项目内部的模块。

为遵守风格指南中的规定，模块名称要短、使用小写，并避免使用特殊符号，比如点(.)和问号(?)。如 `my.spam.py` 这样的名字是必须不能用的！该方式命名将妨碍Python的模块查找功能。就 `my.spam.py` 来说，Python 认为需要在 `my` 文件夹中找到 `spam.py` 文件，实际并不是这样。这个例子example 展示了点表示法应该如何在Python文件中使用。如果愿意您可以将模块命名为 `my_spam.py`，不过并不推荐在模块名中使用下划线。但是，在模块名称中使用其他字符（空格或连字号）将阻止导入（-是减法运算符），因此请尽量保持模块名称简单，以无需分开单词。最重要的是，不要使用下划线命名空间，而是使用子模块。

```
1. # OK
2. import library.plugin.foo
3. # not OK
4. import library.foo_plugin
```

除了以上的命名限制外，Python文件成为模块没有其他特殊的要求，但为了合理地使用这个观念并避免问题，您需要理解`import`的原理机制。具体来说，`import modu` 语句将寻找合适的文件，即调用目录下的 `modu.py` 文件（如果该文件存在）。如果没有找到这份文件，Python解释器递归地在 "PYTHONPATH" 环境变量中查找该文件，如果仍没有找到，将抛出`ImportError`异常。

一旦找到 `modu.py`，Python解释器将在隔离的作用域内执行这个模块。所有顶层语句都会被执行，包括其他的引用。方法与类的定义将会存储到模块的字典中。然后，这个模块的变量、方法和类通过命名空间暴露给调用方，这是Python中特别有用和强大的核心概念。

在很多其他语言中，`include file` 指令被预处理器用来获取文件里的所有代码并‘复制’到调用方的代码中。Python则不一样：`include`代码被独立放在模块命名空间里，这意味着您一般不需要担心`include`的代码可能造成不好的影响，例如重载同名方法。

也可以使用`import`语句的特殊形式 `from modu import` 模拟更标准的行为。但 `import` 通常被认为是不好的做法。使用 `from modu import` 的代码较难阅读而且依赖独立性不足。使用 `from modu import func` 能精确定位您想导入的方法并将其放到全局命名空间中。比 `from modu import` 要好些，因为它明确地指明往全局命名空间中导

入了什么方法，它和 `import modu` 相比唯一的优点是之后使用方法时可以少打点儿字。

差

```
1. [...]
2. from modu import *
3. [...]
4. x = sqrt(4) # sqrt是模块modu的一部分么？或是内建函数么？上文定义了么？
```

稍好

```
1. from modu import sqrt
2. [...]
3. x = sqrt(4) # 如果在import语句与这条语句之间，sqrt没有被重复定义，它也许是模块modu的一部分。
```

最好的做法

```
1. import modu
2. [...]
3. x = modu.sqrt(4) # sqrt显然是属于模块modu的。
```

在 [代码风格](#) 章节中提到，可读性是Python最主要特性之一。可读性意味着避免无用且重复的文本和混乱的结构，因而需要花费一些努力以实现一定程度的简洁。但不能过份简洁而导致简短晦涩。除了简单的单文件项目外，其他项目需要能够明确指出类和方法的出处，例如使用 `modu.func` 语句，这将显著提升代码的可读性和易理解性。

包

Python提供非常简单的包管理系统，即简单地将模块管理机制扩展到一个目录上(目录扩展为包)。

任意包含 `init.py` 文件的目录都被认为是一个Python包。导入一个包里不同模块的方式和普通的导入模块方式相似，特别的地方是 `init.py` 文件将集合所有包范围内的定义。

`pack/` 目录下的 `modu.py` 文件通过 `import pack.modu` 语句导入。该语句会在 `pack` 目录下寻找 `init.py` 文件，并执行其中所有顶层语句。以上操作之后，`modu.py` 内定义的所有变量、方法和类在 `pack.modu` 命名空间中均可看到。

一个常见的问题是往 `init.py` 中加了过多代码，随着项目的复杂度增长，目录结构越来越深，子包和更深嵌套的子包可能会出现。在这种情况下，导入多层嵌套的子包中的某个部件需要执行所有通过路径里碰到的 `init.py` 文件。如果包内的模块和子包没有代码共享的需求，使用空白的 `init.py` 文件是正常甚至好的做法。

最后，导入深层嵌套的包可用这个方便的语法：`import very.deep.module as mod`。该语法允许使用 `mod` 替代冗长的 `very.deep.module`。

面向对象编程

Python有时被描述为面向对象编程的语言，这多少是个需要澄清的误导。在Python中一切都是对象，并且能按对象的方式处理。这么说的意思是，例如函数是一等对象。函数、类、字符串乃至类型都是Python对象：与其他对象一

样，他们有类型，能作为函数参数传递，并且还可能有自己的方法和属性。这样理解的话，Python是一种面向对象语言。

然而，与Java不同的是，Python并没有将面向对象编程作为最主要的编程范式。非面向对象的Python项目(比如，使用较少甚至不使用类定义，类继承，或其它面向对象编程的机制)也是完全可行的。

此外在 [模块](#) 章节里曾提到，Python管理模块与命名空间的方式提供给开发者一个自然的方式以实现抽象层的封装和分离，这是使用面向对象最常见的原因。因而，如果业务逻辑没有要求，Python开发者有更多自由去选择不使用面向对象。

在一些情况下，需要避免不必要的面向对象。当我们想要将状态与功能结合起来，使用标准类定义是有效的。但正如函数式编程所讨论的那个问题，函数式的“变量”状态与类的状态并不相同。

在某些架构中，典型代表是web应用，大量Python进程实例被产生以响应可能同时到达的外部请求。在这种情况下，在实例化对象内保持某些状态，即保持某些环境静态信息，容易出现并发问题或竞态条件。有时候在对象状态的初始化(通常通过 `init()` 方法实现)和在其方法中使用该状态之间，环境发生了变化，保留的状态可能已经过时。举个例子，某个请求将对象加载到内存中并标记它为已读。如果同时另一个请求要删除这个对象，删除操作可能刚好发生在第一个请求加载完该对象之后，结果就是第一个请求标记了一个已经被删除的对象为已读。

这些问题使我们产生一个想法：使用无状态的函数是一种更好的编程范式。另一种建议是尽量使用隐式上下文和副作用较小的函数与程序。函数的隐式上下文由函数内部访问到的所有全局变量与持久层对象组成。副作用即函数可能使其隐式上下文发生改变。如果函数保存或删除全局变量或持久层中数据，这种行为称为副作用。

把有隐式上下文和副作用的函数与仅包含逻辑的函数(纯函数)谨慎地区分开来，会带来以下好处：

- 纯函数的结果是确定的：给定一个输入，输出总是固定相同。
- 当需要重构或优化时，纯函数更易于更改或替换。
- 纯函数更容易做单元测试：很少需要复杂的上下文配置和之后的数据清除工作。
- 纯函数更容易操作、修饰和分发。

总之，对于某些架构而言，纯函数比类和对象在构建模块时更有效率，因为他们没有任何上下文和副作用。但显然在很多情况下，面向对象编程是有用甚至必要的。例如图形桌面应用或游戏的开发过程中，操作的元素(窗口、按钮、角色、车辆)在计算机内存里拥有相对较长的生命周期。

装饰器

Python语言提供一个简单而强大的语法：‘装饰器’。装饰器是一个函数或类，它可以包装(或装饰)一个函数或方法。被‘装饰’的函数或方法会替换原来的函数或方法。由于在Python中函数是一等对象，它也可以被‘手动操作’，但是使用[@decorators](#)语法更清晰，因此首选这种方式。

```

1. def foo():
2.     # 实现语句
3.
4. def decorator(func):
5.     # 操作func语句
6.     return func
7.
8. foo = decorator(foo) # 手动装饰
9.
10. @decorator

```

```

11. def bar():
12.     # 实现语句
13. # bar()被装饰了

```

这个机制对于分离概念和避免外部不相关逻辑“污染”主要逻辑很有用处。记忆化

<<https://en.wikipedia.org/wiki/Memoization#Overview>>; 或缓存就是一个很好的使用装饰器的例子：您需要在table中储存一个耗时函数的结果，并且下次能直接使用该结果，而不是再计算一次。这显然不属于函数的逻辑部分。

上下文管理器

上下文管理器是一个Python对象，为操作提供了额外的上下文信息。这种额外的信息，在使用 `with` 语句初始化上下文，以及完成 `with` 块中的所有代码时，采用可调用的形式。这里展示了使用上下文管理器的为人熟知的示例，打开文件：

```

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

```

任何熟悉这种模式的人都知道以这种形式调用 `open` 能确保 `f` 的 ``close`` 方法会在某个时候被调用。这样可以减少开发人员的认知负担，并使代码更容易阅读。

实现这个功能有两种简单的方法：使用类或使用生成器。让我们自己实现上面的功能，以使用类方式开始：

```

1. class CustomOpen(object):
2.     def __init__(self, filename):
3.         self.file = open(filename)
4.
5.     def __enter__(self):
6.         return self.file
7.
8.     def __exit__(self, ctx_type, ctx_value, ctx_traceback):
9.         self.file.close()
10.
11. with CustomOpen('file') as f:
12.     contents = f.read()

```

这只是一个常规的Python对象，它有两个由 `with` 语句使用的额外方法。`CustomOpen` 首先被实例化，然后调用它的 `__enter__` 方法，而且 `__enter__` 的返回值在 `as f` 语句中被赋给 `f`。当 `with` 块中的内容执行完后，会调用 `__exit__` 方法。

而生成器方式使用了Python自带的`contextlib`：

```

1. from contextlib import contextmanager
2.
3. @contextmanager
4. def custom_open(filename):
5.     f = open(filename)
6.     try:

```

```

7.         yield f
8.     finally:
9.         f.close()
10.
11. with custom_open('file') as f:
12.     contents = f.read()

```

这与上面的类示例道理相通，尽管它更简洁。`customopen` 函数一直运行到 `yield` 语句。然后它将控制权返回给 `with` 语句，然后在 `as f` 部分将 `yield` 的 `_f` 赋值给 `f`。`finally` 确保不论 `with` 中是否发生异常，`close()` 都会被调用。

由于这两种方法都是一样的，所以我们应该遵循Python之禅来决定何时使用哪种。如果封装的逻辑量很大，则类的方法可能会更好。而对于处理简单操作的情况，函数方法可能会更好。

动态类型

Python是动态类型语言，这意味着变量并没有固定的类型。实际上，Python 中的变量和其他语言有很大的不同，特别是静态类型语言。变量并不是计算机内存中被写入的某个值，它们只是指向内存的‘标签’或‘名称’。因此可能存在这样的情况，变量 '`a`' 先代表值1，然后变成字符串 '`a string`'，然后又变为指向一个函数。

Python 的动态类型常被认为是它的缺点，的确这个特性会导致复杂度提升和难以调试的代码。命名为 '`a`' 的变量可能是各种类型，开发人员或维护人员需要在代码中追踪命名，以保证它没有被设置到毫不相关的对象上。

这里有些避免发生类似问题的参考方法：

- 避免对不同类型的对象使用同一个变量名
差

```

1. a = 1
2. a = 'a string'
3. def a():
4.     pass # 实现代码

```

好

```

1. count = 1
2. msg = 'a string'
3. def func():
4.     pass # 实现代码

```

使用简短的函数或方法能降低对不相关对象使用同一个名称的风险。即使是相关的不同类型的对象，也更建议使用不同命名：

差

```

1. items = 'a b c d' # 首先指向字符串...
2. items = items.split(' ') # ...变为列表
3. items = set(items) # ...再变为集合

```

重复使用命名对效率并没有提升：赋值时无论如何都要创建新的对象。然而随着复杂度的提升，赋值语句被其他代码包括 'if' 分支和循环分开，使得更难查明指定变量的类型。在某些代码的做法中，例如函数编程，推荐的是从不重复对同一个变量命名赋值。Java内的实现方式是使用 'final' 关键字。Python并没有 'final' 关键字而且这与它的哲学相悖。尽管如此，避免给同一个变量命名重复赋值仍是个好的做法，并且有助于掌握可变与不可变类型的概念。

可变和不可变类型

Python提供两种内置或用户定义的类型。可变类型允许内容的内部修改。典型的动态类型包括列表与字典：列表都有可变方法，如 `list.append()` 和 `list.pop()`，并且能就地修改。字典也是一样。不可变类型没有修改自身内容的方法。比如，赋值为整数6的变量 `x` 并没有 "自增" 方法，如果需要计算 `x + 1`，必须创建另一个整数变量并给其命名。

```

1. my_list = [1, 2, 3]
2. my_list[0] = 4
3. print my_list # [4, 2, 3] <- 原列表改变了
4.
5. x = 6
6. x = x + 1 # x 变量是一个新的变量

```

这种差异导致的一个后果就是，可变类型是不 '稳定' 的，因而不能作为字典的键使用。合理地使用可变类型与不可变类型有助于阐明代码的意图。例如与列表相似的不可变类型是元组，创建方式为 `(1, 2)`。元组是不可修改的，并能作为字典的键使用。

Python 中一个可能会让初学者惊讶的特性是：字符串是不可变类型。这意味着当需要组合一个字符串时，将每一部分放到一个可变列表里，使用字符串时再组合（'join'）起来的做法更高效。值得注意的是，使用列表推导的构造方式比在循环中调用 `append()` 来构造列表更好也更快。

差

```

1. # 创建将0到19连接起来的字符串 (例 "012..1819")
2. nums = ""
3. for n in range(20):
4.     nums += str(n) # 慢且低效
5. print nums

```

好

```

1. # 创建将0到19连接起来的字符串 (例 "012..1819")
2. nums = []
3. for n in range(20):
4.     nums.append(str(n))
5. print "".join(nums) # 更高效

```

更好

```
1. # 创建将0到19连接起来的字符串 (例 "012..1819")
```

```

2. nums = [str(n) for n in range(20)]
3. print ''.join(nums)

```

最好Best

```

1. # 创建将0到19连接起来的字符串 (例 "012..1819")
2. nums = map(str, range(20))
3. print ''.join(nums)

```

最后关于字符串的说明的一点是，使用 `join()` 并不总是最好的选择。比如当用预先确定数量的字符串创建一个新的字符串时，使用加法操作符确实更快，但在上文提到的情况下或添加到已存在字符串的情况下，使用 `join()` 是更好的选择。

```

1. foo = 'foo'
2. bar = 'bar'
3.
4. foobar = foo + bar # 好的做法
5. foo += 'ooo' # 不好的做法，应该这么做：
6. foo = ''.join([foo, 'ooo'])

```

注解

除了 `str.join()` 和 `+`，您也可以使用 `%` 格式运算符来连接确定数量的字符串，但 [PEP 3101](#) 建议使用 `str.format()` 替代 `%` 操作符。

```

1. foo = 'foo'
2. bar = 'bar'
3.
4. foobar = '%s%s' % (foo, bar) # 可行
5. foobar = '{0}{1}'.format(foo, bar) # 更好
6. foobar = '{foo}{bar}'.format(foo=foo, bar=bar) # 最好

```

提供依赖关系

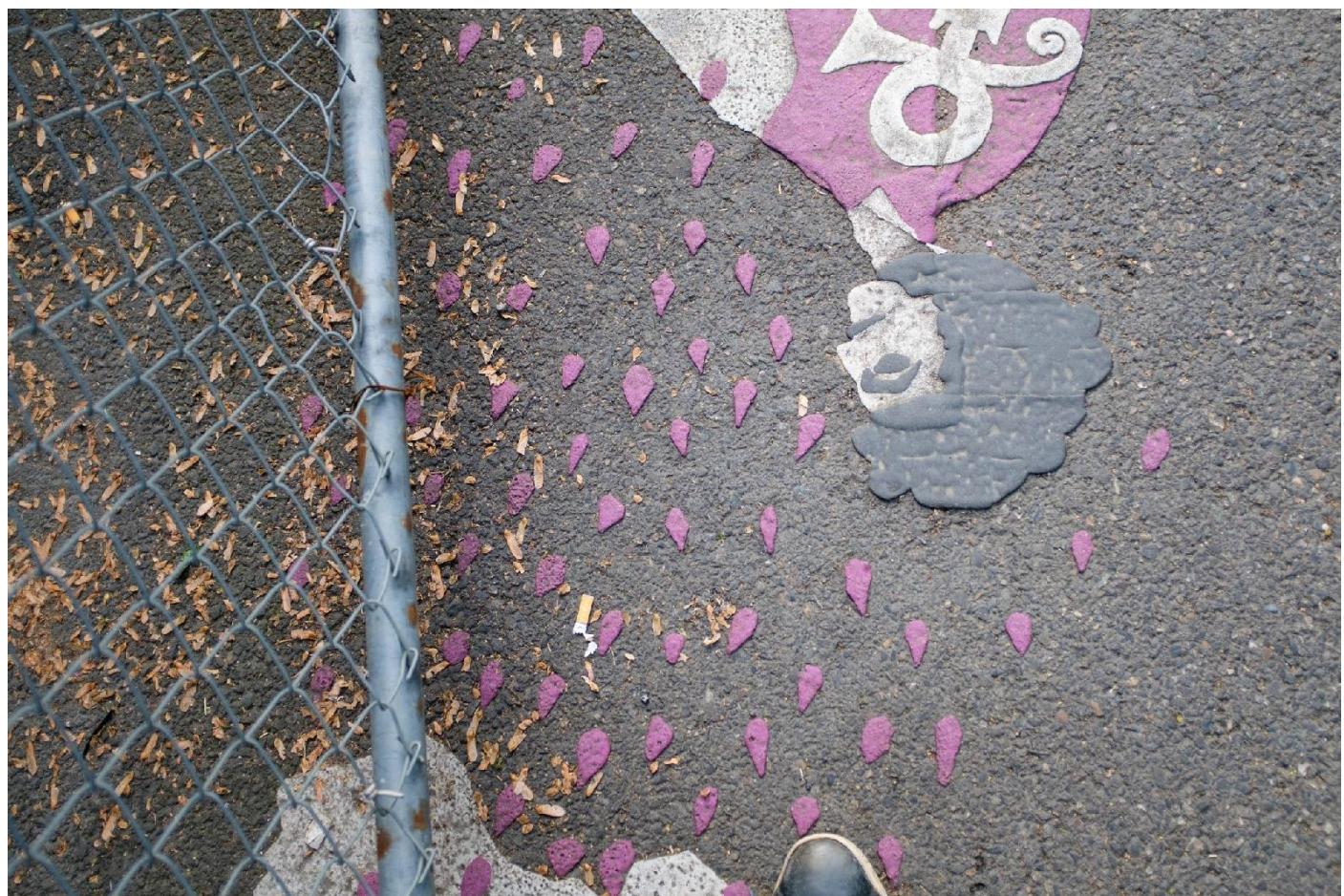
Runners

更多阅读

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

原文：<http://pythonguidecn.readthedocs.io/zh/latest/writing/structure.html>

代码风格



如果您问Python程序员最喜欢Python的什么，他们总会说是Python的高可读性。事实上，高度的可读性是Python语言的设计核心。这基于这样的事实：代码的阅读比编写更加频繁。

Python代码具有高可读性的其中一个原因是它的相对完整的代码风格指引和“*Pythonic*”的习语。

当一位富有经验的Python开发者（Pythonista）指出某段代码并不“*Pythonic*”时，通常意味着这些代码并没有遵循通用的指导方针，也没有用最佳的（最可读的）方式来表达意图。

在某些边缘情况下，Python代码中并没有大家都认同的表达意图的最佳方式，但这些情况都很少见。

一般概念

明确的代码

在存在各种黑魔法的Python中，我们提倡最明确和直接的编码方式。

糟糕

```
1. def make_complex(*args):
2.     x, y = args
3.     return dict(**locals())
```

优雅

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

在上述优雅的代码中，`x`和`y`以明确的字典形式返回给调用者。开发者在使用这个函数的时候通过阅读第一和最后一行，能够正确地知道该做什么。而在糟糕的例子中则没有那么明确。

每行一个声明

复合语句（比如说列表推导）因其简洁和表达性受到推崇，但在同一行代码中写两条独立的语句是糟糕的。

糟糕

```
1. print 'one'; print 'two'
2.
3. if x == 1: print 'one'
4.
5. if <complex comparison> and <other complex comparison>:
6.     # do something
```

优雅

```
1. print 'one'
2. print 'two'
3.
4. if x == 1:
5.     print 'one'
6.
7. cond1 = <complex comparison>
8. cond2 = <other complex comparison>
9. if cond1 and cond2:
10.    # do something
```

函数参数

将参数传递给函数有四种不同的方式：

- 位置参数 是强制的，且没有默认值。 它们是最简单的参数形式，而且能被用在一些这样的函数参数中：它们是函数意义的完整部分，其顺序是自然的。比如说：对函数的使用者而言，记住 `send(message, recipient)` 或 `point(x, y)` 需要两个参数以及它们的参数顺序并不困难。
在这两种情况下，当调用函数的时候可以使用参数名称，也可以改变参数的顺序，比如说 `send(recipient='World', message='Hello')` 和 `point(y=2, x=1)`。但和 `send('Hello', 'World')` 和 `point(1, 2)` 比起来，这降低了可读性，而且带来了不必要的冗长。
- 关键字参数 是非强制的，且有默认值。它们经常被用在传递给函数的可选参数中。当一个函数有超过两个或三个位置参数时，函数签名会变得难以记忆，使用带有默认参数的关键字参数将会带来帮助。比如，一个更完整的 `send` 函数可以被定义为 `send(message, to, cc=None, bcc=None)`。这里的 `cc` 和 `bcc` 是可选的，当

没有传递给它们其他值的时候，它们的值就是None。

Python中有多种方式调用带关键字参数的函数。比如说，我们可以按定义中的参数顺序而无需明确的命名参数来调用函数，就像 `send('Hello', 'World', 'Cthulhu', 'God')` 是将密件发送给上帝。我们也可以使用命名参数而无需遵循参数顺序来调用函数，就像 `send('Hello again', 'World', bcc='God', cc='Cthulhu')` 。如果没有任何强有力的理由不去遵循最接近函数定义的语法： `send('Hello', 'World', cc='Cthulhu', bcc='God')` 那么这两种方式都应该是要极力避免的。

作为附注，请遵循 [YAGNI](#) 原则。通常，移除一个用作“以防万一”但却看起来从未使用的可选参数（以及它在函数中的逻辑），比添加一个所需的新的可选参数和它的逻辑要来的困难。

- 任意参数列表 是第三种给函数传参的方式。如果函数的目的通过带有数目可扩展的位置参数的签名能够更好的表达，该函数可以被定义成 `args` 的结构。在这个函数体中，`args` 是一个元组，它包含所有剩余的位置参数。举个例子，我们可以用任何容器作为参数去调用 `send(message, args)`，比如 `send('Hello', 'God', 'Mom', 'Cthulhu')`。在此函数体中，`args` 相当于 `('God', 'Mom', 'Cthulhu')`。尽管如此，这种结构有一些缺点，使用时应该予以注意。如果一个函数接受的参数列表具有相同的性质，通常把它定义成一个参数，这个参数是一个列表或者其他任何序列会更清晰。在这里，如果 `send` 参数有多个容器（`recipients`），将之定义成 `send(message, recipients)` 会更明确，调用它时就使用 `send('Hello', ['God', 'Mom', 'Cthulhu'])`。这样的话，函数的使用者可以事先将容器列表维护成列表（list）形式，这为传递各种不能被转变成其他序列的序列（包括迭代器）带来了可能。
 - 任意关键字参数字典 是最后一种给函数传参的方式。如果函数要求一系列待定的命名参数，我们可以使用 `**kwargs` 的结构。在函数体中，`kwargs` 是一个字典，它包含所有传递给函数但没有被其他关键字参数捕捉的命名参数。
- 和 任意参数列表 中所需注意的一样，相似的原因是：这些强大的技术是用在被证明确实需要用到它们的时候，它们不应该被用在能用更简单和更明确的结构，来足够表达函数意图的情况下。

编写函数的时候采用何种参数形式，是用位置参数，还是可选关键字参数，是否使用形如任意参数的高级技术，这些都由程序员自己决定。如果能明智地遵循上述建议，就可能且非常享受地写出这样的Python函数：

- 易读（名字和参数无需解释）
- 易改（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

Python 对骇客来说是一个强有力的工具，它拥有非常丰富的钩子（hook）和工具，允许您施展几乎任何形式的技巧。比如说，它能够做以下每件事：

- 改变对象创建和实例化的方式
- 改变Python解释器导入模块的方式
- 甚至可能（如果需要的话也是被推荐的）在Python中嵌入C程序

尽管如此，所有的这些选择都有许多缺点。使用更加直接的方式来达成目标通常是更好的方法。它们最主要的缺点是可读性不高。许多代码分析工具，比如说 `pylint` 或者 `pyflakes`，将无法解析这种“魔法”代码。

我们认为Python开发者应该知道这些近乎无限的可能性，因为它为我们灌输了没有不可能完成的任务的信心。然而，知道如何，尤其是何时 不能 使用它们是非常重要的。

就像一位功夫大师，一个Pythonista知道如何用一个手指杀死对方，但从不会那么去做。

我们都是负责任的用户

如前所述，Python允许很多技巧，其中一些具有潜在的危险。一个好的例子是：任何客户端代码能够重写一个对象的属性和方法（Python中没有“private”关键字）。这种哲学是在说：“我们都是负责任的用户”，它和高度防御性的语言（如Java，拥有很多机制来预防错误的使用）有着非常大的不同。

这并不意味着，比如说，Python中没有属性是私有的，也不意味着没有合适的封装方法。与其依赖在开发者的代码之间树立起的一道道隔墙，Python社区更愿意依靠一组约定，来表明这些元素不应该被直接访问。

私有属性的主要约定和实现细节是在所有的“内部”变量前加一个下划线。如果客户端代码打破了这条规则并访问了带有下划线的变量，那么因内部代码的改变而出现的任何不当的行为或问题，都是客户端代码的责任。

鼓励“慷慨地”使用此约定：任何不开放给客户端代码使用的方法或属性，应该有一个下划线前缀。这将保证更好的职责划分以及更容易对已有代码进行修改。将一个私有属性公开化总是可能的，但是把一个公共属性私有化可能是一个更难的选择。

返回值

当一个函数变得复杂，在函数体中使用多返回值的语句并不少见。然而，为了保持函数的明确意图以及一个可持续的可读水平，更建议在函数体中避免使用返回多个有意义的值。

在函数中返回结果主要有两种情况：函数正常运行并返回它的结果，以及错误的情况，要么因为一个错误的输入参数，要么因为其他导致函数无法完成计算或任务的原因。

如果您在面对第二种情况时不想抛出异常，返回一个值（比如说None或False）来表明函数无法正确运行，可能是需要的。在这种情况下，越早返回所发现的不正确上下文越好。这将帮助扁平化函数的结构：在“因为错误而返回”的语句后的所有代码能够假定条件满足接下来的函数主要结果的运算。有多个这样的返回结果通常是需要的。

尽管如此，当一个函数在其正常过程中有多个主要出口点时，它会变得难以调试和返回其结果，所以保持单个出口点可能会更好。这也将有助于提取某些代码路径，而且多个出口点很有可能意味着这里需要重构。

```

1. def complex_function(a, b, c):
2.     if not a:
3.         return None # 抛出一个异常可能会更好
4.     if not b:
5.         return None # 抛出一个异常可能会更好
6.
7.     # 一些复杂的代码试着用a, b, c来计算x
8.     # 如果成功了，抵制住返回x的诱惑
9.     if not x:
10.        # 一些关于x的计算的Plan-B
11.        return x # 返回值x只有一个出口点有利于维护代码

```

习语 (Idiom)

编程习语，说得简单些，就是写代码的方式。编程习语的概念在 [c2](#) 和 [Stack Overflow](#) 上有充足的讨论。

采用习语的Python代码通常被称为 *Pythonic*。

尽管通常有一种 -- 而且最好只有一种 -- 明显的方式去写得Pythonic；对Python初学者来说，写出习语式的Python代码的方式并不明显。所以，好的习语必须有意识地获取。

如下有一些常见的Python习语：

解包 (Unpacking)

如果您知道一个列表或者元组的长度，您可以将其解包并为它的元素取名。比如，`enumerate()` 会对list中的每个项提供包含两个元素的元组：

```
1. for index, item in enumerate(some_list):
2.     # 使用index和item做一些工作
```

您也能通过这种方式交换变量：

```
1. a, b = b, a
```

嵌套解包也能工作：

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

在Python 3中，扩展解包的新方法在 [PEP 3132](#) 有介绍：

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

创建一个被忽略的变量

如果您需要赋值（比如，在 [解包 \(Unpacking\)](#)）但不需要这个变量，请使用 `_`：

```
1. filename = 'foobar.txt'
2. basename, _, ext = filename.rpartition('.')
```

注解

许多Python风格指南建议使用单下划线的 "`_`" 而不是这里推荐的双下划线 "`__`" 来指示废弃变量。问题是，"`_`" 常用在作为 `gettext()` 函数的别名，也被用在交互式命令行中记录最后一次操作的值。相反，使用双下划线十分清晰和方便，而且能够消除使用其他这些用例所带来的意外干扰的风险。

创建一个含N个对象的列表

使用Python列表中的 `*` 操作符：

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

创建一个含N个列表的列表

因为列表是可变的，所以 `*` 操作符（如上）将会创建一个包含N个且指向 同一个列表的列表，这可能不是您想用的。取而代之，请使用列表解析：

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

注意：在 Python 3 中使用 `range()` 而不是 `xrange()`

根据列表来创建字符串

创建字符串的一个常见习语是在空的字符串上使用 `str.join()`。

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

这会将 `word` 变量赋值为 '`spam`'。这个习语可以用在列表和元组中。

在集合体（collection）中查找一个项

有时我们需要在集合体中查找。让我们看看这两个选择：列表和集合（`set`）。

用如下代码举个例子：

```
1. s = set(['s', 'p', 'a', 'm'])
2. l = ['s', 'p', 'a', 'm']
3.
4. def lookup_set(s):
5.     return 's' in s
6.
7. def lookup_list(l):
8.     return 's' in l
```

即使两个函数看起来完全一样，但因为 `查找集合` 是利用了Python中的集合是可哈希的特性，两者的查询性能是非常不同的。为了判断一个项是否在列表中，Python将会查看每个项直到它找到匹配的项。这是耗时的，尤其是对长列表而言。另一方面，在集合中，项的哈希值将会告诉Python在集合的哪里去查找匹配的项。结果是，即使集合很大，查询的速度也很快。在字典中查询也是同样的原理。想了解更多内容，请见[StackOverflow](#)。想了解在每种数据结构上的多种常见操作的花费时间的详细内容，请见 [此页面](#)。

因为这些性能上的差异，在下列场合在使用集合或者字典而不是列表，通常会是个好主意：

- 集合体中包含大量的项
- 您将在集合体中重复地查找项
- 您没有重复的项

对于小的集合体，或者您不会频繁查找的集合体，建立哈希带来的额外时间和内存的开销经常会大过改进搜索速度所节省的时间。

Python之禅

又名 **PEP 20**, Python设计的指导原则。

```

1. >>> import this
2. The Zen of Python, by Tim Peters
3.
4. Beautiful is better than ugly.
5. Explicit is better than implicit.
6. Simple is better than complex.
7. Complex is better than complicated.
8. Flat is better than nested.
9. Sparse is better than dense.
10. Readability counts.
11. Special cases aren't special enough to break the rules.
12. Although practicality beats purity.
13. Errors should never pass silently.
14. Unless explicitly silenced.
15. In the face of ambiguity, refuse the temptation to guess.
16. There should be one-- and preferably only one --obvious way to do it.
17. Although that way may not be obvious at first unless you're Dutch.
18. Now is better than never.
19. Although never is often better than *right* now.
20. If the implementation is hard to explain, it's a bad idea.
21. If the implementation is easy to explain, it may be a good idea.
22. Namespaces are one honking great idea -- let's do more of those!
23.
24. Python之禅 by Tim Peters
25.
26. 优美胜于丑陋 (Python以编写优美的代码为目标)
27. 明了胜于晦涩 (优美的代码应当是明了的, 命名规范, 风格相似)
28. 简洁胜于复杂 (优美的代码应当是简洁的, 不要有复杂的内部实现)
29. 复杂胜于凌乱 (如果复杂不可避免, 那代码间也不能有难懂的关系, 要保持接口简洁)
30. 扁平胜于嵌套 (优美的代码应当是扁平的, 不能有太多的嵌套)
31. 间隔胜于紧凑 (优美的代码有适当的间隔, 不要奢望一行代码解决问题)
32. 可读性很重要 (优美的代码是可读的)
33. 即便假借特例的实用性之名, 也不可违背这些规则 (这些规则至高无上)
34. 不要包容所有错误, 除非您确定需要这样做 (精准地捕获异常, 不写 except:pass 风格的代码)
35. 当存在多种可能, 不要尝试去猜测
36. 而是尽量找一种, 最好是唯一一种明显的解决方案 (如果不确定, 就用穷举法)
37. 虽然这不容易, 因为您不是 Python 之父 (这里的 Dutch 是指 Guido )
38. 做也许好过不做, 但不假思索就动手还不如不做 (动手之前要细思量)
39. 如果您无法向人描述您的方案, 那肯定不是一个好方案; 反之亦然 (方案测评标准)
40. 命名空间是一种绝妙的理念, 我们应当多加利用 (倡导与号召)

```

想要了解一些Python优雅风格的例子, 请见 [这些来自于Python用户的幻灯片](#).

PEP 8

PEP 8 是Python事实上的代码风格指南, 我们可以在 pep8.org 上获得高质量的、一度的PEP 8版本。

强烈推荐阅读这部分。整个Python社区都尽力遵循本文档中规定的准则。一些项目可能受其影响，而其他项目可能修改其建议。

也就是说，让您的 Python 代码遵循 PEP 8 通常是个好主意，这也有助于在与其他开发人员一起工作时使代码更加具有可持续性。命令行程序 `pycodestyle` <https://github.com/PyCQA/pycodestyle>（以前叫做 `pep8`），可以检查代码一致性。在您的终端上运行以下命令来安装它：

```
1. $ pip install pycodestyle
```

然后，对一个文件或者一系列的文件运行它，来获得任何违规行为的报告。

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

程序 `autopep8` 能自动将代码格式化成 PEP 8 风格。用以下指令安装此程序：

```
1. $ pip install autopep8
```

用以下指令格式化一个文件：

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

不包含 `-in-place` 标志将会使得程序直接将更改的代码输出到控制台，以供审查。`-aggressive` 标志则会执行更多实质性的变化，而且可以多次使用以达到更佳的效果。

约定

这里有一些您应该遵循的约定，以让您的代码更加易读。

检查变量是否等于常量

您不需要明确地比较一个值是True，或者None，或者0 - 您可以仅仅把它放在if语句中。参阅 [真值测试](#) 来了解什么被认为是false。

糟糕：

```
1. if attr == True:
2.     print 'True!'
3.
4. if attr == None:
```

```
5.     print 'attr is None!'
```

优雅：

```
1. # 检查值
2. if attr:
3.     print 'attr is truthy!'
4.
5. # 或者做相反的检查
6. if not attr:
7.     print 'attr is falsey!'
8.
9. # or, since None is considered false, explicitly check for it
10. if attr is None:
11.     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` 语法，或者将一个默认参数传递给 `dict.get()`。

糟糕：

```
1. d = {'hello': 'world'}
2. if d.has_key('hello'):
3.     print d['hello']    # 打印 'world'
4. else:
5.     print 'default_value'
```

优雅：

```
1. d = {'hello': 'world'}
2.
3. print d.get('hello', 'default_value') # 打印 'world'
4. print d.get('thingy', 'default_value') # 打印 'default_value'
5.
6. # Or:
7. if 'hello' in d:
8.     print d['hello']
```

维护列表的捷径

[列表推导](#)提供了一个强大的而又简洁的方式来处理列表。而且，`map()` 和 `filter()` 函数用一种不同且更简洁的语法处理列表。

糟糕：

```
1. # 过滤大于 4 的元素
2. a = [3, 4, 5]
```

```

3. b = []
4. for i in a:
5.     if i > 4:
6.         b.append(i)

```

优雅：

```

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

```

糟糕：

```

1. # 所有的列表成员都加 3
2. a = [3, 4, 5]
3. for i in range(len(a)):
4.     a[i] += 3

```

优雅：

```

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

```

使用 `enumerate()` 获得列表中的当前位置的计数。

```

1. a = [3, 4, 5]
2. for i, item in enumerate(a):
3.     print i, item
4. # 打印
5. # 0 3
6. # 1 4
7. # 2 5

```

使用 `enumerate()` 函数比手动维护计数有更好的可读性。而且，它对迭代器进行了更好的优化。

读取文件

使用 `with open` 语法来读取文件。它将会为您自动关闭文件。

糟糕：

```

1. f = open('file.txt')
2. a = f.read()
3. print a
4. f.close()

```

优雅：

```

1. with open('file.txt') as f:
2.     for line in f:
3.         print line

```

`with` 语句会更好，因为它能确保您总是关闭文件，即使是在 `with` 的区块中抛出一个异常。

行的延续

当一个代码逻辑行的长度超过可接受的限度时，您需要将之分为多个物理行。如果行的结尾是一个反斜杠（），Python解释器会把这些连续行拼接在一起。这在某些情况下很有帮助，但我们总是应该避免使用，因为它的脆弱性：如果在行的结尾，在反斜杠后加了空格，这会破坏代码，而且可能有意想不到的结果。

一个更好的解决方案是在元素周围使用括号。左边以一个未闭合的括号开头，Python解释器会把行的结尾和下一行连接起来直到遇到闭合的括号。同样的行为适用中括号和大括号。

糟糕：

```

1. my_very_big_string = """For a long time I used to go to bed early. Sometimes, \
2.     when I had put out my candle, my eyes would close so quickly that I had not even \
3.         time to say "I'm going to sleep.""""
4.
5. from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
6.     yet_another_nice_function

```

优雅：

```

1. my_very_big_string = (
2.     "For a long time I used to go to bed early. Sometimes, "
3.     "when I had put out my candle, my eyes would close so quickly "
4.     "that I had not even time to say "I'm going to sleep."
5. )
6.
7. from some.deep.module.inside.a.module import (
8.     a_nice_function, another_nice_function, yet_another_nice_function)

```

尽管如此，通常情况下，必须去分割一个长逻辑行意味着您同时想做太多的事，这可能影响可读性。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/writing/style.html>

阅读好的代码



设计Python的核心理念是创建可读性代码。这种设计的目的非常简单：Python编写者的首要事情是阅读好的代码。

成为优秀Python编写的秘诀是去阅读，理解和领会好的代码。

良好的代码通常遵循[代码风格](#) 中的指南，尽可能向读者表述地简洁清楚。

以下是推荐阅读的Python项目。每个项目都是Python代码的典范。

- [Howdoi](#)Howdoi是代码搜寻工具，使用Python编写。
- [Flask](#)Flask是基于Werkzeug和Jinja2，使用Python的微框架。它能够快速启动，并且开发意图良好。
- [Diamond](#)Diamond是python的守护进程，它收集指标，并且将他们发布至Graphite或其它后端。它能够收集cpu, 内存，网络，i/o，负载和硬盘指标。除此，它拥有实现自定义收集器的API，该API几乎能从任何资源中获取指标。
- [Werkzeug](#)Werkzeug起初只是一个WSGI应用多种工具的集成，现在它已经变成非常重要的WSGI实用模型。它包括强大的调试器，功能齐全的请求和响应对象，处理entity tags的HTTP工具，缓存控制标头，HTTP数据，cookie处理，文件上传，强大的URL路由系统和一些社区提供的插件模块。
- [Requests](#)Requests是Apache2许可的HTTP库，使用Python编写。
- [Tablib](#)Tablib是无格式的表格数据集库，使用Python编写。

待处理

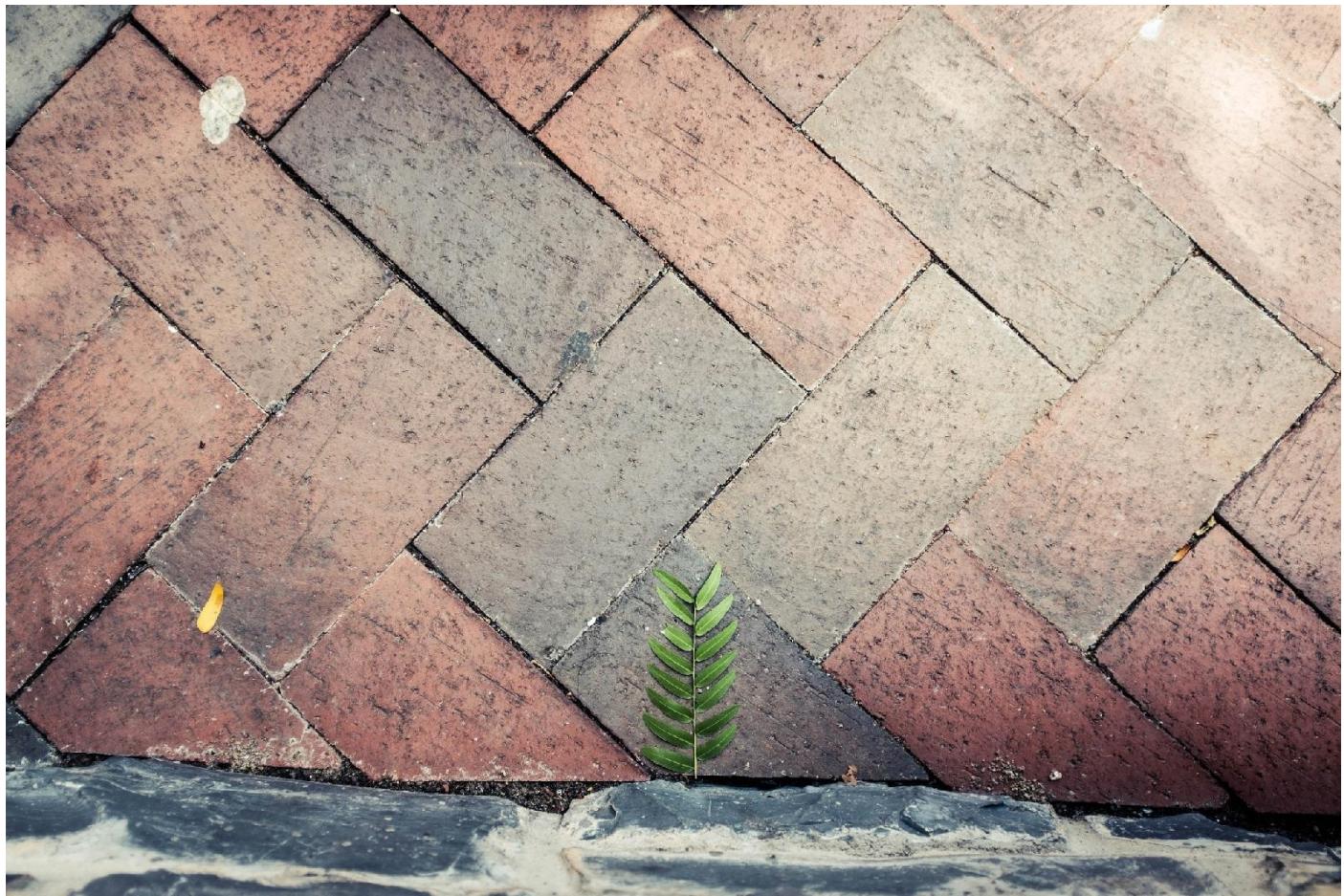
包括每个列出项目中典型代码的例子。解释为什么它是非常优秀的代码，举出较复杂的例子。

待处理

解释快速识别数据结构，算法，并确定代码内容的技术。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/writing/reading.html>

文档



可读性是Python开发者需关注的重点，这包括项目和代码文档的可读性。遵循一些简单的最佳实践能够帮助您和其他人节省很多时间。

项目文档

根目录下的 `README` 文件应该告诉用户和维护者关于项目的基本信息。它应该是原始文本，用一些非常容易阅读的标记，比如 `reStructuredText` 或 `Markdown`。它应该包含几行内容用来解释项目或者库的目的（假设用户不用知道任何关于该项目的事），软件主要源的URL，以及一些基本的信用信息。此文件是代码阅读者的主要入口。

`INSTALL` 文件对Python来说并不必要。安装指令通常少至一条命令，比如说 `pip install module` 或 `python setup.py install`，这被添加到 `README` 文件中。

`LICENSE` 文件应该 总是 存在的，而且应该指定向公众开放的软件许可。

`TODO` 文件或者位于 `README` 中的 `TODO` 区域应该列出代码的开发计划。

`CHANGELOG` 文件或者位于 `README` 中的 `CHANGELOG` 区域应该呈现对代码库的最新修改的简短概述。

项目发布

根据项目，您的文档中可能包含下列部分或所有的内容：

- 一份 **介绍** 应该用一两个极其简化的用例，来非常简短地概述该产品能够用来做什么。这是您的项目的30秒的自我陈述 (*thirty-second pitch*)。
- 一份 **教程** 应该展示一些主要的用例，它要有更多细节。读者将会跟着一步步搭建工作原型。
- 一份 **API 参考** 通常从代码（参见 [docstrings](#)）中产生。它会列出所有的可供公共访问的接口、参数和返回值。
- **开发人员文档** 适用于潜在贡献者。这可以包括项目的代码惯例和通用设计策略。

Sphinx

[Sphinx](#) 无疑是最流行的Python文档工具。请使用它吧。 它能把 [reStructuredText](#) 标记语言转换为广泛的输出格式，包括HTML、LaTeX（可打印PDF版本）、手册页和纯文本。

[Read The Docs](#) 是一个 超棒的 并且 免费的 文档托管，可以托管您的 [Sphinx](#) 文档。请使用它。您可以为它配置提交钩子到您的源库中，这样文档的重新构建将会自动进行。

运行时，[Sphinx](#) 将导入您的代码，并使用Python的内省功能，它将提取所有函数，方法和类签名。它还将提取附带的文档字符串，并将其全部编译成结构良好且易于阅读的文档。

注解

[Sphinx](#) 因API生成而著名，但它也适用于普通的项目文档。本指南使用 [Sphinx](#) 进行构建，并托管在 [Read The Docs](#) 上。

reStructuredText

大多数Python文档是用 [reStructuredText](#) 编写的。它就像是内建了所有可选扩展的Markdown。

[reStructuredText Primer](#) 和 [reStructuredText Quick Reference](#) 应该会帮助您熟悉它的语法。

代码文档建议

注释能使代码清晰，将其加入到代码中是为了理解代码起来更容易。在Python中，注意以一个hash（数字符号）（“#”）开始。

在Python中， 文档字符串 用来描述模块、类和函数：

```
1. def square_and_rooter(x):
2.     """返回自己乘以自己的平方根。"""
3.     ...
```

一般来说，要遵循 [PEP 8#comments](#)（“Python风格指南”）的注释部分。更多关于文档字符串的内容可以在 [PEP 0257#specification](#)（文档字符串约定指引）上找到。

注释代码块

不要使用三引号去注释代码。 这不是好的实践，因为面向行的命令行工具，比如说grep，不会知道注释过的代码是没有激活的。对每一个注释行，使用带有合适缩进的井号会更好。您的编辑器可能很容易做到这一点，并能切换注释/

取消注释。

文档字符串和魔法

一些工具使用文档字符串来嵌入不止是文档的行为，比如说单元测试逻辑。这些可能不错，但是简单地“保持文档就是文档”您永远都不会错。

像 [Sphinx](#) 这样的工具会将您的文档字符串解析为reStructuredText，并以HTML格式正确呈现。这使得在示例代码片段中嵌入项目的文档非常简单。

此外，[Doctest](#) 将读取所有内嵌的看起来像Python命令行输入（以“>>>”为前缀）的文档字符串，并运行，以检查命令输出是否匹配其下行内容。这允许开发人员在源码中嵌入真实的示例和函数的用法。此外，它还能确保代码被测试和工作。

```

1. def my_function(a, b):
2.     """
3.     >>> my_function(2, 3)
4.     6
5.     >>> my_function('a', 3)
6.     'aaa'
7.     """
8.     return a * b

```

文档字符串 vs 块注释

这些不可互换。对于函数或类，开头的注释区是程序员的注解。而文档字符串描述了函数或类的 操作：

```

1. # 由于某种原因这个函数减慢程序执行。
2. def square_and_rooter(x):
3.     """返回自己乘以自己的平方根。"""
4.     ...

```

与块注释不同，文档字符串内置于Python语言本身。与被优化掉的注释相比较，这意味着您可以使用Python强大的内省功能以在运行时获得文档字符串。对于几乎每个Python对象，可以通过其 `doc` 属性或使用内置的“`help()`”函数访问文档字符串。

块注释通常用于解释一段代码是 做什么，或是算法的细节。而文档字符串更适合于向其他用户（或是写完代码6个月内的您）解释您代码中的特定功能是 如何 使用，或是方法、类和模块的作用。

编写文本字符串

取决于函数、方法或类的复杂度，使用单行文档字符串可能十分合适。以下通常用于非常明显的情况，例如：

```

1. def add (a, b) :
2.     """两个数字相加，并返回结果。””
3.     return a + b

```

文档字符串应该以易于理解的方式描述函数。对于简单的例子，如简单的函数和类，简单地将函数的签名（即 `b) -> result`）嵌入到文档字符串中是不必要的。这是因为使用Python的“inspect”模块可以很容易地找到这些信息。此外，通过阅读源代码也可以很容易地获得。

然而，在更大或更复杂的项目中，提供相关功能的更多信息是个好主意，包括它是做什么的，所抛的任何异常，返回的内容或参数的相关细节。

对于更详细的代码文档，用于Numpy项目的风格较为流行，通常称为 [Numpy style](#) 文档字符串。虽然它之前的例子可能会占用更多的行，但它允许开发人员包含方法、函数或类的更多信息。

```

1. def random_number_generator (arg1, arg2) :
2.     """
3.     摘要行。
4.
5.     扩展功能描述。
6.
7.     参数
8.     -----
9.     arg1 : int
10.    arg1的描述
11.    arg2 : str
12.    arg2的描述
13.
14.    返回
15.    -----
16.    int
17.    返回值说明
18.
19.    """
20.    return 42

```

`sphinx.ext.napoleon` 插件可以让Sphinx解析这种风格的文档字符串，使您可以轻松地将NumPy风格的文档输入到项目中。

最后，编写文档字符串的风格并没那么重要，它们的目的是为任何可能需要阅读或更改代码的人提供文档。只要它是正确的，可以理解的，切中相关点，那么它就完成了所设计的工作。

要进一步阅读docstrings，请随时参见 [PEP 257](#)

其他工具

您可能在其他场景看到过这些。使用 [Sphinx](#)。

Pycco

Pycco是一个“文学编程风格的文档生成器”，它是node.js [Docco](#) 的移植版本。它将代码生成为一个并排的HTML代码和文档。

Ronn

Ronn用来构建Unix手册。它将人可读的文本文件转换成用于终端显示的roff文件，以及用于web的HTML文件。

Epydoc

Epydoc已经中断开发。使用 [Sphinx](#) 来替代。

MkDocs

MkDocs是一个快速简单的静态网站生成器，它适合于构建使用Markdown的项目文档。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/writing/documentation.html>

测试您的代码



测试您的代码非常重要。

常常将测试代码和运行代码一起写是一种非常好的习惯。聪明地使用这种方法将会帮助您更加精确地定义代码的含义，并且代码的耦合性更低。

测试的通用规则：

- 测试单元应该集中于小部分的功能，并且证明它是对的。
- 每个测试单元必须完全独立。他们都能够单独运行，也可以在测试套件中运行，而不用考虑被调用的顺序。要想实现这个规则，测试单元应该加载最新的数据集，之后再做一些清理。这通常用方法 `setUp()` 和 `tearDown()` 处理。
- 尽量使测试单元快速运行。如果一个单独的测试单元需要较长的时间去运行，开发进度将会延迟，测试单元将不能如期常态化运行。有时候，因为测试单元需要复杂的数据结构，并且当它运行时每次都要加载，所以其运行时间较长。把运行吃力的测试单元放在单独的测试组件中，并且按照需要运行其它测试单元。
- 学习使用工具，学习如何运行一个单独的测试用例。然后，当在一个模块中开发了一个功能时，经常运行这个功能的测试用例，理想情况下，一切都将自动。
- 在编码会话前后，要常常运行完整的测试组件。只有这样，您才会坚信剩余的代码不会中断。
- 实现钩子（hook）是一个非常好的主意。因为一旦把代码放入分享仓库中，这个钩子可以运行所有的测试单元。
- 如果您在开发期间不得不打断自己的工作，写一个被打断的单元测试，它关于下一步要开发的东西。当回到工作时，您将更快地回到原先被打断的地方，并且步入正轨。

- 当您调试代码的时候，首先需要写一个精确定位bug的测试单元。尽管这样做很难，但是捕捉bug的单元测试在项目中很重要。
- 测试函数使用长且描述性的名字。这边的样式指导与运行代码有点不一样，运行代码更倾向于使用短的名字，而测试函数不会直接被调用。在运行代码中，`square()`或者甚至`sqr()`这样的命名都是可以的，但是在测试代码中，您应该这样取名`test_square_of_number_2()`, `test_square_negative_number()`。当测试单元失败时，函数名应该显示，而且尽可能具有描述性。
- 当发生了一些问题，或者不得不改变时，如果代码中有一套不错的测试单元，维护将很大一部分依靠测试组件解决问题，或者修改确定的行为。因此测试代码应该尽可能多读，甚至多于运行代码。目的不明确的测试单元在这种情况下没有多少用处。
- 测试代码的另外一个用处是作为新开发人员的入门介绍。当有人需要基于现有的代码库工作时，运行并且阅读相关的测试代码是最好的做法。他们会或者应该发现大多数困难出现的热点，以及边界的情况。如果他们必须添加一些功能，第一步应该是添加一个测试，以确保新的功能不是一个尚未插入到界面的工作路径。

基本

单元测试

`unittest` 包括Python标准库中的测试模型。任何一个使用过Junit, nUnit, 或CppUnit工具的人对它的API都会比较熟悉。

创建测试用例通过继承 `unittest.TestCase` 来实现。

```

1. import unittest
2.
3. def fun(x):
4.     return x + 1
5.
6. class MyTest(unittest.TestCase):
7.     def test(self):
8.         self.assertEqual(fun(3), 4)

```

因为Python 2.7单元测试也包括自己的发现机制。

[在标准库文档中单元测试](#)

文档测试

`doctest` 模块查找零碎文本，就像在Python中`docstrings`内的交互式会话，执行那些会话以证实工作正常。

`doctest`模块的用例相比之前的单元测试有所不同：它们通常不是很详细，并且不会用特别的用例或者处理模糊的回归bug。作为模块和其部件主要用例的表述性文档，`doctest`模块非常有用。

函数中的一个简单的`doctest`:

```

1. def square(x):
2.     """返回 x 的平方。
3.

```

```

4.     >>> square(2)
5.     4
6.     >>> square(-2)
7.     4
8.     """
9.
10.    return x * x
11.
12. if __name__ == '__main__':
13.     import doctest
14.     doctest.testmod()

```

当使用 `python module.py` 这样的命令行运行这个模块时, `doctest` 将会运行, 并会在结果不和文档字符串的描述一致时报错。

工具

py.test

相比于Python标准的单元测试模块, `py.test` 是一个没有模板的选择。

```
1. $ pip install pytest
```

尽管这个测试工具功能完备, 并且可扩展, 但是它语法很简单。创建一个测试组件和写一个带有诸多函数的模块一样容易:

```

1. # content of test_sample.py
2. def func(x):
3.     return x + 1
4.
5. def test_answer():
6.     assert func(3) == 5

```

运行命令`py.test`

```

1. $ py.test
2. ===== test session starts =====
3. platform darwin -- Python 2.7.1 -- pytest-2.2.1
4. collecting ... collected 1 items
5.
6. test_sample.py F
7.
8. ===== FAILURES =====
9. _____ test_answer _____
10.
11. def test_answer():
12. >     assert func(3) == 5
13. E     assert 4 == 5

```

测试您的代码

```
14. E           +   where 4 = func(3)
15.
16. test_sample.py:5: AssertionError
17. ===== 1 failed in 0.02 seconds =====
```

要比单元测试模型中相同功能所要求的工作量少得多。

py.test

Hypothesis

Hypothesis 让您编写被示例源码参数化的测试的库。它会生成简单易懂的例子，使您的测试失败，让您花更少的力气找到更多的错误。

```
1. $ pip install hypothesis
```

例如，测试浮动列表要尝试很多例子，但是会报告每个错误的最小例子（区分异常类型和位置）：

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

```
1. Falsifying example: test_mean(
2.     xs=[1.7976321109618856e+308, 6.102390043022755e+303]
3. )
```

Hypothesis 是实用的，也是非常强大的，能经常会找出被其他所有形式的测试所遗漏的错误。它能与py.test很好地集成，在简单和高级场景中都非常注重可用性。

hypothesis

tox

tox是自动化测试管理和针对多种解释器配置测试工具。

```
1. $ pip install tox
```

tox允许通过简单的初始化样式配置文件，配置复杂的多参数测试矩阵。

tox

Unittest2

Unittest2是Python2.7中unittest模型的补丁，它的API有所改善，并且对Python之前版本中已有的内容有了更好的说明。

如果使用Python2.6版本或者以下，需要使用pip安装unittest2。

```
1. $ pip install unittest2
```

将来您可能想要以unittest之名导入模块，目的是更容易地把代码移植到新的版本中。

```
1. import unittest2 as unittest
2.
3. class MyTest(unittest.TestCase):
4.     ...
```

如果切换到新的Python版本，并且不再需要unittest2模块，您只需要在测试模块中改变import内容，而不必改变其它代码。

unittest2

mock

`unittest.mock` 是Python中用于测试的一个库。在Python3.3版本中，标准库中就有。[标准库](#)。

对于Python相对早的版本，如下操作：

```
1. $ pip install mock
```

在测试环境下，使用mock对象能够替换部分系统，并且对它们如何被使用做了声明。例如，您可以对一个方法打猴子补丁：

例如，您可以对一个方法打猴子补丁：

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

在测试环境下，对于模型中的mock类或对象，使用补丁修饰器。在下面这个例子中，一直返回相同结果的外部查询系统使用mock替换（但仅用在测试期间）。

```
1. def mock_search(self):
2.     class MockSearchQuerySet(SearchQuerySet):
3.         def __iter__(self):
4.             return iter(["foo", "bar", "baz"])
5.     return MockSearchQuerySet()
6.
7. # SearchForm here refers to the imported class reference in myapp,
8. # not where the SearchForm class itself is imported from
9. @mock.patch('myapp.SearchForm.search', mock_search)
```

```
10. def test_new_watchlist_activities(self):  
11.     # get_search_results runs a search and iterates over the result  
12.     self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

mock有许多其它方法，您可以配置它，并且控制它的动作。

mock

原文: <http://pythonguidecn.readthedocs.io/zh/latest/writing/tests.html>

日志 (Logging)



日志模块自2.3版本开始便是Python标准库的一部分。它被简洁的描述在 [PEP 282](#)。众所周知，除了 [基础日志指南](#) 部分，该文档并不容易阅读。

日志的两个目的：

- **诊断日志** 记录与应用程序操作相关的日志。例如，用户遇到的报错信息，可通过搜索诊断日志获得上下文信息。
- **审计日志** 为商业分析而记录的日志。从审计日志中，可提取用户的交易信息，并结合其他用户资料构成用户报告或者用来优化商业目标。

... 或者打印？

当需要在命令行应用中显示帮助文档时，[打印](#) 是一个相对于日志更好的选择。而在其他时候，日志总能优于 [打印](#)，理由如下：

- 日志事件产生的 [日志记录](#)，包含清晰可用的诊断信息，如文件名称、路径、函数名和行号等。
- 包含日志模块的应用，默认可通过根记录器对应用的日志流进行访问，除非您将日志过滤了。
- 可通过 `logging.Logger.setLevel()` 方法有选择地记录日志，或可通过设置 `logging.Logger.disabled` 属性为 `True` 来禁用。

库中的日志

[日志指南](#) 中含 [库日志配置](#) 的说明。由于是 用户，而非库来指明如何响应日志事件，因此这里有一个值得反复说明的忠告：

注解

强烈建议不要向您的库日志中加入除NullHandler外的其它处理程序。

在库中，声明日志的最佳方式是通过 `name` 全局变量： `logging` 模块通过点 (dot) 运算符创建层级排列的日志，因此，用 `name` 可以避免名字冲突。

以下是一个来自 [requests 资源](#) 的最佳实践的例子 — 把它放置在您的 `init.py` 文件中

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

应用程序中的日志

应用程序开发的权威指南， [应用的12要素](#)，也在其中一节描述了[日志的作用](#)。它特别强调将日志视为事件流，并将其发送至由应用环境所处理的标准输出中。

配置日志至少有以下三种方式：

- 使用INI格式文件：
 - 优点：使用 `logging.config.listen()` 函数监听socket，可在运行过程中更新配置
 - 缺点：通过源码控制日志配置较少（例如 子类化定制的过滤器或记录器）。
- 使用字典或JSON格式文件：
 - 优点：除了可在运行时动态更新，在Python 2.6之后，还可通过 `json` 模块从其它文件中导入配置。
 - 缺点：很难通过源码控制日志配置。
- 使用源码：
 - 优点：对配置绝对的控制。
 - 缺点：对配置的更改需要对源码进行修改。

通过INI文件进行配置的例子

我们假设文件名为 `logging_config.ini`。关于文件格式的更多细节，请参见[日志指南](#) 中的 [日志配置](#) 部分。

```
1. [loggers]
2. keys=root
3.
4. [handlers]
5. keys=stream_handler
6.
7. [formatters]
8. keys=formatter
9.
10. [logger_root]
```

```

11. level=DEBUG
12. handlers=stream_handler
13.
14. [handler_stream_handler]
15. class=StreamHandler
16. level=DEBUG
17. formatter=formatter
18. args=(sys.stderr,)
19.
20. [formatter_formatter]
21. format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s

```

然后在源码中调用 `logging.config.fileConfig()` 方法：

```

1. import logging
2. from logging.config import fileConfig
3.
4. fileConfig('logging_config.ini')
5. logger = logging.getLogger()
6. logger.debug('often makes a very good meal of %s', 'visiting tourists')

```

通过字典进行配置的例子

Python 2.7中，您可以使用字典实现详细配置。PEP 391 包含了一系列字典配置的强制和非强制的元素。

```

1. import logging
2. from logging.config import dictConfig
3.
4. logging_config = dict(
5.     version = 1,
6.     formatters = {
7.         'f': {'format':
8.             '%(asctime)s%(name)-12s%(levelname)-8s%(message)s'
9.         },
10.    },
11.    handlers = {
12.        'h': {'class': 'logging.StreamHandler',
13.              'formatter': 'f',
14.              'level': logging.DEBUG}
15.    },
16.    root = {
17.        'handlers': ['h'],
18.        'level': logging.DEBUG,
19.    },
20.
21. dictConfig(logging_config)
22.
23. logger = logging.getLogger()
24. logger.debug('often makes a very good meal of %s', 'visiting tourists')

```

通过源码直接配置的例子

```
1. import logging
2.
3. logger = logging.getLogger()
4. handler = logging.StreamHandler()
5. formatter = logging.Formatter(
6.     '%(asctime)s%(name)-12s%(levelname)-8s%(message)s')
7. handler.setFormatter(formatter)
8. logger.addHandler(handler)
9. logger.setLevel(logging.DEBUG)
10.
11. logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

原文: <http://pythonguidecn.readthedocs.io/zh/latest/writing/logging.html>

常见陷阱



大多数情况下，Python的目标是成为一门简洁和一致的语言，同时避免意外情况。然而，有些情况可能会使新人困惑。

其中一些情况是有意为之的，但可能有潜在的风险。而另一些可以说是语言的缺陷。总的来说，下面是一些乍看起来很取巧的行为，不过只要您注意了强调的事项，这些行为通常是可取的。

可变默认参数

看起来，最让Python程序员感到惊奇的是Python对函数定义中可变默认参数的处理。

您所写的

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

您所期望的

```
1. my_list = append_to(12)
2. print(my_list)
```

```

3.
4. my_other_list = append_to(42)
5. print(my_other_list)

```

每次调用函数时，如果不提供第二个参数，就会创建一个新的列表，所以结果应是这样的：

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

而事实是

```

1. [12]
2. [12, 42]

```

当函数被定义时，一个新的列表就被创建一次，而且同一个列表在每次成功的调用中都被使用。

当函数被定义时，Python的默认参数就被创建一次，而不是每次调用函数的时候创建。这意味着，如果您使用一个可变默认参数并改变了它，您 将会 在未来所有对此函数的调用中改变这个对象。

您应该做的

在每次函数调用中，通过使用指示没有提供参数的默认参数（`None` 通常是个好选择），来创建一个新的对象。

```

1. def append_to(element, to=None):
2.     if to is None:
3.         to = []
4.     to.append(element)
5.     return to

```

别忘了，您在把 `列表` 对象作为第二个参数传入。

什么情况下陷阱不是陷阱

有时您可以专门“利用”（或者说特地使用）这种行为来维护函数调用间的状态。这通常用于编写缓存函数。

迟绑定闭包

另一个常见的困惑是Python在闭包（或在周围全局作用域（surrounding global scope））中绑定变量的方式。

您所写的

```

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

```

您所期望的

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

一个包含五个函数的列表，每个函数有它们自己的封闭变量 `i` 乘以它们的参数，得到：

```
1. 0
2. 2
3. 4
4. 6
5. 8
```

而事实是

```
1. 8
2. 8
3. 8
4. 8
5. 8
```

五个函数被创建了，它们全都用4乘以 `x`。

Python的闭包是 **迟绑定**。这意味着闭包中用到的变量的值，是在内部函数被调用时查询得到的。

这里，不论 任何 返回的函数是如何被调用的，`i` 的值是调用时在周围作用域中查询到的。接着，循环完成，`i` 的值最终变成了4。

关于这个陷阱有一个普遍严重的误解，它被认为是和Python的**Lambdas** 有关。由 `lambda` 表达式创建的函数并没什么特别，而且事实上，同样的问题也出现在使用普通的 `def` 定义 上：

```
1. def create_multipliers():
2.     multipliers = []
3.
4.     for i in range(5):
5.         def multiplier(x):
6.             return i * x
7.         multipliers.append(multiplier)
8.
9.     return multipliers
```

您应该做的

最一般的解决方案可以说是有点取巧 (hack)。由于Python拥有在前文提到的为函数默认参数赋值的行为（参见 [可变默认参数](#)），您可以创建一个立即绑定参数的闭包，像下面这样：

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

或者，您可以使用 `functools.partial` 函数：

```
1. from functools import partial
2. from operator import mul
3.
4. def create_multipliers():
5.     return [partial(mul, i) for i in range(5)]
```

什么情况下陷阱不是陷阱

有时您就想要闭包有如此表现，迟绑定在很多情况下是不错的。不幸的是，循环创建独特的函数是一种会使它们出差错的情况。

字节码（.pyc）文件无处不在！

默认情况下，当使用文件执行Python代码时，Python解释器会自动将该文件的字节码版本写入磁盘。比如，`module.pyc`。

这些“.pyc”文件不应该加入到您的源代码仓库。

理论上，出于性能原因，此行为默认为开启。没有这些字节码文件，Python会在每次加载文件时重新生成字节码。

禁用字节码（.pyc）文件

幸运的是，生成字节码的过程非常快，在开发代码时不需要担心。

那些文件很讨厌，所以让我们摆脱他们吧！

```
1. $ export PYTHONDONTWRITEBYTECODE=1
```

使用 `$PYTHONDONTWRITEBYTECODE` 环境变量，Python则不会把这些文件写入磁盘，您的开发环境将会保持良好和干净。

我建议在您的 `~/.profile` 里设置这个环境变量。

删除字节码（.pyc）文件

以下是删除所有已存在的字节码文件的好方法：

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

从项目根目录运行，所有 `.pyc` 文件会嗖地一下消失，好多了~

版本控制忽略

如果由于性能原因仍然需要 `.pyc` 文件，您可以随时将它们添加到版本控制存储库的忽略文件中。流行的版本控制

系统能够使用文件中定义的通配符来应用特殊规则。

一份忽略文件将确保匹配的文件未被检入存储库。Git 使用 `.gitignore`，而 Mercurial 使用 ```.hgignore```。

至少您的忽略文件应该是这样的。

```
1. syntax:glob    # .gitignore 文件不需要这行  
2. *.py[cod]      # 将匹配 .pyc、.pyo 和 .pyd文件  
3. __pycache__/  # 排除整个文件夹
```

您可能希望根据需要添加更多文件和目录。下次提交到存储库时，这些文件将不被包括。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/writing/gotchas.html>

选择一个许可



源代码发布 需要 一个许可。在美国，如果没有指定许可，用户就无权下载、修改或分发。此外，人们无法贡献代码，除非您告诉他们所遵守的规则。选择许可是复杂的，这里有一些指导方针：

开源。这里有很多 [开源许可](#)可以选择。

一般来说，这些许可大致分为两类：

- 许可更关注用户随意使用软件的自由（较宽松的自由软件开源许可，如 MIT、BSD，以及 Apache）。
- 许可更关注确保代码 – 包括对其任意的修改和分发 – 的自由（较不宽松的自由软件许可，如GPL 和 LGPL）。

后者相较而言不太宽松，它们不允许他人在软件中添加代码，也不允许分发软件包括对其源代码的更改。

为了帮助您选择用于项目的许可，这里有一个 [许可选择器](#)，可供使用。

较宽松：

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

较不宽松：

- LGPL
- GPL
 - GPLv2
 - GPLv3

关于许可中使用软件时什么能做、不能做、必须做的解释，这里 [tldrLegal](#) 有很好的概述。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/writing/license.html>

- 网络应用
- Web 应用 & 框架
- HTML 抓取
- 命令行应用
- GUI应用
- 数据库
- 网络
- 系统管理
- 持续集成
- 速度
- 科学应用
- 图像处理
- 数据序列化
- XML解析
- JSON
- 密码学
- 机器学习
- 与C/C++库交互

网络应用



HTTP

超文本传输协议(HTTP)是为分布式的、协同的多媒体信息系统而设计的应用协议，是万维网数据传输的基础。

Requests

Python的urllib2标准模块涵盖了所需的大多数HTTP功能，但它的API却是支离破碎的。它构建在一个和现今完全不同的时期—以及为了一个不一样的网络。一个简单的任务便需要耗费它大量的工作(即使重写函数也无济于事)。

Requests将所有Python HTTP相关的功能剥离了出来，并与网络服务无缝衔接。Requests无需再在URL中添加查询语句或格式编码的POST数据。而集成在Requests中urllib3，则实现了持久连接和HTTP连接池的完全自动化。

- [文档](#)
- [PyPi](#)
- [GitHub](#)

分布式系统

ZeroMQ

ØMQ(也被称为ZeroMQ, ØMQ 或 ZMQ)是一种高性能异步消息传递库，旨在应用于可扩展分布的或并发的应用。它提供一个消息队列，但与面向消息的中间件不同，ØMQ系统可在不依赖专用消息代理的情况下运行。ØMQ旨在设计成为类似于socket风格的API。

RabbitMQ

RabbitMQ是一种使用了高级消息队列协议(AMQP)的开源消息代理软件。RabbitMQ服务由Erlang编程语言写成，并构建在开放电信平台框架上，应用于集群和故障转移。与该代理交互的客户端库支持所有主流编程语言。

- [主页](#)
- [GitHub组织](#)

原文: <http://pythonguidecn.readthedocs.io/zh/latest/scenarios/client.html>

Web 应用 & 框架



Python作为一门强大的脚本语言，能够适应快速原型和较大项目的制作，它被广泛用于web应用程序的开发中。

Context

WSGI

Web服务网关接口 (Web Server Gateway Interface, 简称“WSGI”) 是一种在Web服务器和Python Web应用程序或框架之间的标准接口。通过标准化Web服务器和Python web应用程序或框架之间的行为和通信，WSGI使得编

写可移植的的Python web代码变为可能，使其能够部署在任何
上。WSGI记录在 [PEP 3333](#)。

符合WSGI的web服务器 [<wsgi-servers-ref>](#)

框架

广义地说，Web框架包含一系列库和一个主要的处理器 (handler)，这样您就能够构建自己的代码来实现Web应用（比如说一个交互式的网站）。大多数web框架包含模式和工具，至少实现以下功能：

URL路由 (URL Routing)

将输入的HTTP请求匹配到特定的Python代码用来调用

请求和响应对象 (*Request and Response Objects*)

封装来自或发送给用户浏览器的信息

模板引擎 (*Template Engine*)

能够将实现应用的Python代码逻辑和其要产生输出的HTML（或其他）分离开

Web服务器开发 (*Development Web Server*)

在开发机上运行HTTP服务器，从而快速开发；当文件更新时自动更新服务端代码。

Django

[Django](#) 是一个功能齐备的web应用框架。它是创建面向内容网站的极佳选择。通过提供众多工具和模式，Django使得快速构建复杂的、有数据库支持的web应用成为可能，同时鼓励使用它作为编写代码的最佳实践。

Django拥有非常庞大和活跃的社区。此外，许多预构建的 [可重用模块](#)可以原样合并到新工程中，或者定制成符合需求的样子。

在 [美国](#)、[欧洲](#) 和[Australia](#) 均有每年度的Django会议。

如今大部分新的Python web应用都是用Django构建的。

Flask

[Flask](#) 是一款针对Python的“微型框架”，它是构建更小应用、API和web服务的极佳选择。使用Flask构建应用，除了一些函数附上路由，它和写标准Python模块很相似。它真的很赞。

Flask不会提供一切您可能需要的内容，而是实现了web应用框架中最常用的核心组件，比如说URL路由、请求和响应对象和模板等。

作为Flask的用户，由您来决定选择和集成其他您可能用到的组件。比如说数据库访问或者表单生成和验证就不是Flask内置的功能。

这挺好的，因为很多web应用并不需要这些特性。对于那些需要的，有许多可用的 [扩展](#) 或许符合您的需求。

Flask是任何不适用Django的Python web应用的默认选择。

Falcon

当您的目标是构建快速、可扩展的REST风格API微服务时，[Falcon](#) 是个不错的选择。

这是一个可靠的、高性能的Python Web框架，用于构建大规模应用后端和微服务。Falcon鼓励REST架构风格的URI到资源的映射，以花费尽可能少的精力同时又保持高效。

Falcon重点关注四个方面：速度、可靠性、灵活性和可调试性。它通过“响应者（responder）”（诸如 `on_get()`、`on_put()` 等）来实现HTTP。这些响应者接收直接的请求，以及响应对象。

Tornado

[Tornado](#) 是一个面向Python的异步web框架，它有自己的事件。这就使得它，举个例子，可以原生地支持

WebSockets。编写良好的Tornado应用具有卓越的性能特性。

除非您认为自己需要它，我并不建议您使用Tornado。

Pyramid

[Pyramid](#) 是一个非常灵活的框架，重点关注模块化。 它内置少量库（“电池”），并鼓励用户扩展其基本功能。它提供一组cookiecutter模板，帮助用户生成新项目。Pyramid驱动了Python基础架构中最重要部分之一—— [PyPI](#)。

Pyramid不像Django和Flask，并没有庞大的用户基数。它一个有能力的框架，但如今新Python web 应用程序并没有十分青睐它。

Web 服务端

Nginx

[Nginx](#) （发音为 "engine-x"）是一个web服务器，并是HTTP、SMTP和其他协议的反向代理。它由其高性能、相对简洁以及对众多应用服务器（比如WSGI服务器）兼容而著名。它也拥有便利的特性，比如负载均衡、基本的认证、流等。Nginx被设计为承载高负载的网站，并逐渐变得广为流行。

WSGI 服务器

独立WSGI服务器相比传统web服务器，使用更少的资源，并提供最高的性能 [\[1\]](#)。

Gunicorn

[Gunicorn](#) (Green Unicorn, 绿色独角兽) 是一个纯Python WSGI服务器，用来支持Python应用。不像其他Python web服务器，它有周全的用户界面，十分易于使用和配置。

Gunicorn具有合理的默认配置。 然而，其他一些像uWSGI这样的服务器相较而言过于可定制化，因此更加难以高效使用。

Gunicorn是如今新Python web应用程序的推荐选择。

Waitress

[Waitress](#) 是一个纯Python WSGI服务器，声称具备“非常可接受的性能”。它的文档不是很详细，但它确实提供了一些很好的而Gunicorn没有的功能（例如HTTP请求缓冲）。

Waitress在Python Web开发社区中越来越受欢迎。

uWSGI

[uWSGI](#) 用来构建全栈式的主机服务。除了进程管理、进程监控和其他功能外，uWSGI也能作为一个应用服务器，适用于多种编程语言和协议 - 包括Python和WSGI。uWSGI既能当作独立的web路由器来运行，也能运行在一个完整web服务器（比如Nginx或Apache）之后。对于后者，web服务器可以基于 [uwsgi 协议](#) 配置uWSGI和应用的操作。

uwsgi的web服务器支持允许动态配置Python、传递环境变量以及进一步优化。要看更多细节，请看 [uwsgi 魔法变量](#)。

除非您认为自己需要它，我并不建议您使用uwsgi。

服务端最佳实践

如今，自承载Python应用的主体托管于WSGI服务器（比如说 [Gunicorn](#)）或是直接或间接在轻量级web服务器（比如说 [nginx](#)）之后。

WSGI服务器为Python应用服务，它能更好的处理诸如静态文件服务、请求路由、DDoS保护和基本认证的任务。

Hosting

平台即服务（Platform-as-a-Service, PaaS）是一种云计算基础设施类型，抽象和管理基础设施、路由和网络应用的扩展。使用PaaS时，应用开发者只需关注编写应用代码，而无须关心配置细节。

Heroku

[Heroku](#) 为Python 2.7-3.5 应用程序提供一流的支持。

[Heroku](#) 支持所有类型的Python web应用、服务器和框架。在[Heroku](#)上可以免费开发应用程序。

一旦您的应用程序准备好面向生产环境，您可以升级到Hobby或专业应用。

Heroku 维护了使用Python和Heroku交互的 [详细文章](#)，同时也有 [手把手指导](#)来告诉您如何建立第一个应用。

Heroku是如今部署Python Web应用程序的推荐PaaS。

Eldarion

[Eldarion <http://eldarion.cloud/>](#) （被称为Gondor）是由Kubernetes、CoreOS和Docker提供的PaaS。它们支持任一WSGI应用程序，并提供了部署 Django项目 的指南。

模板

多数WSGI应用响应HTTP请求，从而服务于HTML或其他标记语言中的内容。关注点分离的概念建议我们使用模板，而不是直接由Python生成文本内容。模板引擎管理一系列的模板文件，其系统的层次性和包容性避免了不必要的重复。模板引擎负责渲染（产生）实际内容，用由应用生成的动态内容填充静态内容。

由于模板文件有时是由设计师或者前端开发者编写，处理不断增长的复杂度会变得困难。

一些通用的良好实践应用到了部分应用中，情景包括传递动态内容到模板引擎和模板自身中。

- 模板文件只应传递需要渲染的动态内容。避免传递附加的“以防万一”的内容：需要时添加遗漏的变量比移除可能不用的变量要来的容易。
- 许多模板引擎允许在模板中编写复杂语句或者赋值，也有许多允许一些Python代码在模板中等价编写。这种便

利会导致复杂度不可控地增加，也使得查找bug变得更加困难。

- 我们常常需要混合JavaScript模板和HTML模板。一种聪明的做法是孤立出HTML模板传递部分变量内容到JavaScript代码中的部分。

Jinja2

[Jinja2](#) 是一个很受欢迎的模板引擎。

它使用基于文本的模板语言，因此可以用于生成任何类型的标记，而不仅仅是HTML。它允许自定义过滤器，标签，测试和全局变量。它具有Django模板系统的许多改进。

这里有一些在Jinja2中重要的html标签：

```

1. {# 这是注释 #}
2.
3. {# 下一个标签是输出变量 : #}
4. {{title}}
5.
6. {# 区块标签，能通过继承其他html代码来替换区块内容 #}
7. {% block head %}
8. <h1>This is the head!</h1>
9. {% endblock %}
10.
11. {# 数组迭代输出 #}
12. {% for item in list %}
13. <li>{{ item }}</li>
14. {% endfor %}

```

下面列举的内容是一个使用Tornado的站点的例子。Tornado用起来并没那么复杂。

```

1. # 导入 Jinja
2. from jinja2 import Environment, FileSystemLoader
3.
4. # 导入 Tornado
5. import tornado.ioloop
6. import tornado.web
7.
8. # 载入模板文件 templates/site.html
9. TEMPLATE_FILE = "site.html"
10. templateLoader = FileSystemLoader( searchpath="templates/" )
11. templateEnv = Environment( loader=templateLoader )
12. template = templateEnv.get_template(TEMPLATE_FILE)
13.
14. # 包含著名电影的list
15. movie_list = [[1, "The Hitchhiker's Guide to the Galaxy"], [2, "Back to future"], [3, "Matrix"]]
16.
17. # template.render() 返回包含渲染后html的字符串
18. html_output = template.render(list=movie_list,
19.                               title="Here is my favorite movie list")
20.
21. # 主页的handler

```

```

22. class MainHandler(tornado.web.RequestHandler):
23.     def get(self):
24.         # Returns rendered template string to the browser request
25.         self.write(html_output)
26.
27. # 将handler赋给服务器root (127.0.0.1:PORT/)
28. application = tornado.web.Application([
29.     (r"/", MainHandler),
30. ])
31. PORT=8884
32. if __name__ == "__main__":
33.     # Setup the server
34.     application.listen(PORT)
35.     tornado.ioloop.IOLoop.instance().start()

```

`base.html` 文件能够作为所有站点页面的基础，下面是实现的例子。

```

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <html xmlns="http://www.w3.org/1999/xhtml">
4. <head>
5.     <link rel="stylesheet" href="style.css" />
6.     <title>{{title}} - My Webpage</title>
7. </head>
8. <body>
9. <div id="content">
10.    {# 下一行内容将会在site.html模板中被添加。 #}
11.    {% block content %}{% endblock %}
12. </div>
13. <div id="footer">
14.    {% block footer %}
15.    &copy; Copyright 2013 by <a href="http://domain.invalid/">you</a>.
16.    {% endblock %}
17. </div>
18. </body>

```

接下来是我们的site页面（`site.html`），它由Python app载入，并扩展了 `base.html`。下面的内容区块会自动填充到 `base.html` 页面的相关区块中。

```

1. {% extends "base.html" %}
2. {% block content %}
3.     <p class="important">
4.     <div id="content">
5.         <h2>{{title}}</h2>
6.         <p>{{ list_title }}</p>
7.         <ul>
8.             {% for item in list %}
9.                 <li>{{ item[0] }} : {{ item[1] }}</li>
10.            {% endfor %}
11.        </ul>
12.    </div>

```

```

13.      </p>
14.  {% endblock %}

```

Jinja2是新Python Web应用程序的推荐模板库。

Chameleon

[Chameleon](#) 页面模板是使用模板属性语言 (Template Attribute Language, TAL)、[TAL表达语法 \(TAL Expression Syntax, TALES\)](#) 和宏扩展TAL (Macro Expansion TAL, Metal) 语法的HTML/XML模板引擎实现。

Chameleon在Python2.5及以上版本 (包括3.x和pypy) 都是可用的，并常被 [Pyramid Framework](#) 使用。

页面模板是在文档结构中添加特定元素属性和文本标记。使用一系列简单语言概念，您能够控制文档流程、元素重复、文本替换和翻译。由于使用了基于属性的语法，未渲染的页面模板是合法的HTML，它可以在浏览器中查看，甚至能够在WYSIWYG编辑器中编辑。这使得设计者和原型构建者之间在浏览器中静态文件上的往复合作变得更加简单。

从下面的例子中能很快学到基本的TAL语言的用法：

```

1.  <html>
2.   <body>
3.     <h1>Hello, <span tal:replace="context.name">World</span>!</h1>
4.     <table>
5.       <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
6.         <td tal:repeat="col 'juice', 'muffin', 'pie'">
7.           <span tal:replace="row.capitalize()" /> <span tal:replace="col" />
8.         </td>
9.       </tr>
10.      </table>
11.    </body>
12.  </html>

```

用作文本插入的 `` 形式非常常见。如果在未渲染的模板中并不要求严格的合法性，您可以取而代之地使用更加简洁和可读的语法，它使用 `${expression}` 的形式，就像下面这样：

```

1.  <html>
2.   <body>
3.     <h1>Hello, ${world}!</h1>
4.     <table>
5.       <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
6.         <td tal:repeat="col 'juice', 'muffin', 'pie'">
7.           ${row.capitalize()} ${col}
8.         </td>
9.       </tr>
10.      </table>
11.    </body>
12.  </html>

```

但是请记住，全部的 `Default Text`语法也允许在未渲染的模板

中有默认内容。

在来自Pyramid的世界中，Chameleon不被广泛使用。

Mako

Mako 是一种模板语言，为了最大的性能，它编译为了Python。它的语法和API借鉴了其他模板语言，如Django和Jinja2中最好的部分。它是包括 Pylons 和 Pyramid 在内的web框架所使用的默认模板语言。

Mako的一个模板例子如下：

```
1. <%inherit file="base.html"/>
2. <%
3.     rows = [[v for v in range(0,10)] for row in range(0,10)]
4. %>
5. <table>
6.     % for row in rows:
7.     ${makerow(row)}
8.     % endfor
9. </table>
10.
11. <%def name="makerow(row)">
12.     <tr>
13.         % for name in row:
14.             <td>${name}</td>\n
15.         % endfor
16.     </tr>
17. </%def>
```

要渲染一个非常基本的模板，您可以像下面这么做：

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

Mako在Python web社区中受到重视。

References

[1] Benchmark of Python WSGI Servers

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/web.html>

HTML 抓取



Web 抓取

Web站点使用HTML描述，这意味着每个web页面是一个结构化的文档。有时从中获取数据同时保持它的结构是有用的。web站点不总是以容易处理的格式，如 `csv` 或者 `json` 提供它们的数据。

这正是web抓取出场的时机。Web抓取是使用计算机程序将web页面数据进行收集并整理成所需格式，同时保存其结构的实践。

lxml和Requests

`lxml` 是一个优美的扩展库，用来快速解析XML以及HTML文档即使所处理的标签非常混乱。我们也将使用 `Requests` 模块取代内建的`urllib2`模块，因为其速度更快而且可读性更好。您可以通过使用 `pip install lxml` 与 `pip install requests` 命令来安装这两个模块。

让我们以下面的导入开始：

```
1. from lxml import html  
2. import requests
```

下一步我们将使用 `requests.get` 来从web页面中取得我们的数据，通过使用 `html` 模块解析它，并将结果保存到 `tree` 中。

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

`tree` 现在包含了整个HTML文件到一个优雅的树结构中，我们可以使用两种方法访问：XPath以及CSS选择器。在这个例子中，我们将选择前者。

XPath是一种在结构化文档（如HTML或XML）中定位信息的方式。一个关于XPath的不错的介绍参见 [W3Schools](#)。

有很多工具可以获取元素的XPath，如Firefox的FireBug或者Chrome的Inspector。如果您使用Chrome，您可以右键元素，选择 'Inspect element'，高亮这段代码，再次右击，并选择 'Copy XPath'。

在进行一次快速分析后，我们看到在页面中的数据保存在两个元素中，一个是title是'buyer-name' 的div，另一个class是 'item-price' 的span：

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

知道这个后，我们可以创建正确的XPath查询并且使用lxml的 `xpath` 函数，像下面这样：

```
1. #这将创建buyers的列表：
2. buyers = tree.xpath('//div[@title="buyer-name"]/text()')
3. #这将创建prices的列表：
4. prices = tree.xpath('//span[@class="item-price"]/text()')
```

让我们看看我们得到了什么：

```
1. print 'Buyers: ', buyers
2. print 'Prices: ', prices
```

```
1. Buyers: ['Carson Busses', 'Earl E. Byrd', 'Patty Cakes',
2. 'Derri Anne Connecticut', 'Moe Dessa', 'Leda Doggslife', 'Dan Druff',
3. 'Al Fresco', 'Ido Hoe', 'Howie Kisses', 'Len Lease', 'Phil Meup',
4. 'Ira Pent', 'Ben D. Rules', 'Ave Sectomy', 'Gary Shattire',
5. 'Bobbi Soks', 'Sheila Taky', 'Rose Tattoo', 'Moe Tell']
6.
7. Prices: ['$29.95', '$8.37', '$15.26', '$19.25', '$19.25',
8. '$13.99', '$31.57', '$8.49', '$14.47', '$15.86', '$11.11',
9. '$15.98', '$16.27', '$7.50', '$50.85', '$14.26', '$5.68',
10. '$15.00', '$114.07', '$10.09']
```

恭喜！我们已经成功地通过lxml与Request，从一个web页面中抓取了所有我们想要的数据。我们将它们以列表的形式存在内存中。现在我们可以对它做各种很酷的事情了：我们可以使用Python分析它，或者我们可以将之保存为一个文件并向世界分享。

我们可以考虑一些更酷的想法：修改这个脚本来遍历该例数据集中剩余的页面，或者使用多线程重写这个应用从而提

升它的速度。

原文: <http://pythonguidecn.readthedocs.io/zh/latest/scenarios/scrape.html>

命令行应用



注解

想了解关于编写测试的建议, 请查阅 [测试您的代码](#)。

命令行应用, 也被称为 [控制台应用](#)是面向如 `shell`) 之类文本接口的计算机程序。命令行应用通常接收一些输入作为参数, 这些参数 (arguments) 通常被称为参数 (parameters) 或子命令, 而选项 (options) 则被称为flags或switches。

一些流行的命令行应用包括:

- [Grep](#) - 一个纯文本数据搜索工具
- [curl](#) - 基于URL语法的数据传输工具
- [httpie](#) - 一个用户友好的命令行HTTP客户端, 可以代替curl
- [git](#) - 一个分布式版本控制系统
- [mercurial](#) - 一个主体是Python的分布式版本控制系统

Clint

[clint](#) 是一个Python模块, 它包含了很多对命令行应用开发有用的工具。它支持诸如CLI着色以及缩进, 简洁而强大的列打印, 基于进度条的迭代以及参数控制的特性。

Click

`click` 是一个以尽可能少的代码，用组合方式创建命令行接口的Python包。命令行接口创建工具（“Command-line Interface Creation Kit”， Click）高度可配置，但也有开箱即用的默认值设置。

docopt

`docopt` 是一个轻量级，高度Pythonic风格的包，它支持简单而直觉地创建命令行接口，它是通过解析POSIX-style的用法指示文本实现的。

Plac

`Plac` Python标准库`argparse` 的简单封装，它隐藏了大量声明接口的细节：参数解析器是被推断的，其优于写命令明确处理。这个模块的面向是不想太复杂的用户，程序员，系统管理员，科学家以及只是想写个只运行一次的脚本的人们，使用这个命令行接口的理由是它可以快速实现并且简单。

Cliff

`Cliff` 是一个建立命令行程序的框架。它使用`setuptools`入口点（entry points）来提供子命令，输出格式化，以及其他扩展。这个框架可以用来创建多层命令程序，如`subversion`与`git`，其主程序要进行一些简单的参数解析然后调用一个子命令干活。

Cement

`Cement` 是一个高级的CLI应用程序框架。其目标是为简单和复杂的命令行应用程序引入标准和功能完整的平台，并支持快速开发需求，而不会牺牲质量。`Cement`是灵活的，它的用例范围涵盖了从微框架的简单到巨型框架的复杂。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/cli.html>

GUI应用



以下GUI应用按字母顺序排列。

Camelot

除了Python, SQLAlchemy和Qt之外, Camelot提供构建应用的组件。它的灵感取决于Django的管理接口。

如下是主要的信息资源网站: <http://www.python-camelot.com>, 邮件列表网站:

<http://groups.google.com/forum/#!forum/project-camelot>

Cocoa

注解

Cocoa框架仅在OS X中可用。如果您正在写一个跨平台的应用, 不要选用这个!

GTK

PyGTK提供GTK+工具包的Python绑定。和GTK+库一样, PyGTK当前由GNU LGPL授权。PyGTK当前只支持Gtk-2.X的API (Gtk-3.0不支持)。值得一提的是, PyGTK不适用于新项目, 而已经存在的应用不能从PyGTK移植到PyGObject。

PyGObject aka (PyGi)

[PyGObject](#) 提供了Python绑定，可以访问整个GNOME软件平台。它完全兼容GTK+ 3。使用“Python GTK+ 3 教程”(<https://python-gtk-3-tutorial.readthedocs.io/en/latest/>)来开始学习。

[API 参考](#)

Kivy

[Kivy](#) 是一个Python库，该库用于开发多点触控的媒体应用。当代码需要重复利用并且可部署时，它能够实现快速简单的交互设计，并且加速成形。

Kivy使用Python编写，并且基于OpenGL，除此，它支持不同的输入设备，例如鼠标、双鼠标、WiiMote、WM_TOUCH、HIDtouch和苹果的产品等等。

Kivy由社区积极开发，并且免费使用。它适用于所有主要的平台 (Linux, OSX, Windows, Android)

如下是主要的信息资源网站：<http://kivy.org>

PyObjC

注解

仅在os X中可用。如果您正在写一个跨平台的应用，不要选用这个。

PySide

PySide是跨平台的Qt GUI工具包的Python绑定。

```
pip install pyside
```

:Downloads"><https://wiki.qt.io/Category:Downloads>

PyQt

注解

如果软件不能完全遵循GPL，那么就需要商业许可证。

PyQt提供Qt框架的Python绑定（见如下）

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

PyjamasDesktop (pyjs Desktop)

PyjamasDesktop是Pyjamas的端口。PyjamasDesktop是桌面应用工具集，并且是跨平台框架（在发布的v0.6版本之后，PyjamasDesktop是Pyjamas（Pyjs）的一部分）。简而言之，它允许完全一样的Python网页应用资源代码能够如独立的桌面应用执行。

[PyjamasDesktop的Python维基百科](#).

主要的网址：[pyjs Desktop](#).

Qt

Qt 是跨平台应用框架，它被广泛用于借GUI开发软件，但是也可用于非GUI应用。

Toga

Toga 是一个原生Python和操作系统的跨平台GUI工具包。Toga由一个具有共享接口的基础组件库组成，以简化与平台无关的GUI开发。

Toga适用于Mac OS、Windows、Linux（GTK）以及Android和iOS等移动平台。

Tk

Tkinter是Tcl/Tk上的面向对象层。它的优势是包括Python标准库，能够使编程更加方便，兼容性更强。

不管是Tk还是Tkinter，在大多数Unix平台，以及Windows和Macintosh系统都可用。从8.0发布版本开始，Tk在所有平台上使本身的样子和感觉更赞。

在 [TkDocs](#) 中有一个非常好的多语言Tk教程，所有例子使用Python。更多信息可以看 [Python 维基百科](#).

wxPython

wxPython是Python语言编写的GUI工具包。Python编写人员能够使简单容易地使用健壮，高功能的图形用户接口编程。把流行的wxWidgets包在跨平台GUI库中，从而作为Python的扩展模块，这用C++编写。

安装（稳定版）wxPython可以去如下网址<http://www.wxpython.org/download.php#stable>，并且下载适用于当前操作系统的安装包。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/gui.html>

数据库



DB-API

Python数据库API (DB-API) 定义了一个Python数据库访问模块的标准接口。它的文档在 [PEP 249](#) 可以查看。几乎所有Python数据库模块，诸如 `sqlite3`, `psycopg` 以及 `mysql-python` 都遵循这个接口。

关于如何与遵守这一接口的模块交互的教程可以在这里找到：[这里](#) 以及[这里](#) 。

SQLAlchemy

[SQLAlchemy](#) 是一个流行的数据库工具。不像很多数据库库，它不仅提供一个ORM层，而且还有一个通用API来编写避免SQL的数据库无关代码。

```
1. $ pip install sqlalchemy
```

Records

[Records](#) 是极简SQL库，旨在将原始SQL查询发送到各种数据库。数据可以以编程方式使用，也可以导出到一些有用的数据格式。

```
1. $ pip install records
```

还包括用于导出SQL数据的命令行工具。

Django ORM

Django ORM 是 [Django](#) 用来进行数据库访问的接口。

它的思想建立在 [models](#) , 之上。这是一个致力于简化Python中数据操作的抽象层。

基础：

- 每个model是django.db.models.Model的子类。
- model的每个属性表示数据库的域 (field)。
- Django给您一个自动生成的数据库访问API, 参见[Making queries](#)。

peewee

peewee 是另一个ORM, 它致力于轻量级和支持Python2.6+与3.2+默认支持的SQLite, MySQL以及Postgres。[model layer](#)与Django ORM类似并且它拥有 [SQL-like methods](#)来查询数据。除了将SQLite, MySQL以及Postgres变为开箱即用, 还有进一步的扩展功能可以在这里找到: [collection of add-ons](#)。

PonyORM

PonyORM 是一个ORM, 它使用与众不同的方法查询数据库, 有别于使用类似SQL的语言或者布尔表达式, 它使用Python的生成器达到目的。而且还有一个图形化schema编辑器生成PonyORM实体。它支持Python2.6+与3.3+并且可以连接SQLite, MySQL, Postgres与Oracle。

SQLObject

SQLObject 是另一个ORM。它支持广泛的数据库, 常见的MySQL, Postgres以及SQLite与更多的特别系统如SAP DB, SyBase与MSSQL。它只支持Python 2

原文: <http://pythonguidecn.readthedocs.io/zh/latest/scenarios/db.html>

网络



Twisted

[Twisted](#) 是一款基于事件驱动的网络引擎框架。支持创建基于不同网络协议的应用，包括 http 服务器与客户端，SMTP 应用，POP3，IMAP 或 SSH 协议，即时通讯 等等。

PyZMQ

[PyZMQ](#) 是 [ZeroMQ](#) 的 Python 捆绑库 (binding)，ZeroMQ 是一款高效率的异步消息库，它的一个显著优点就是能被用做消息队列且不需要消息代理。基本模式包括：

- 请求-回应模式：连接多个客户端与多个服务端，是远程过程调用和任务分配的模式。
 - 发布-订阅模式：连接多个发布者和多个订阅者，是数据分配模式。
 - 推-挽模式（或管道模式）：连接多个步骤，包括循环的输入/输出端，是并行任务分配和收集模式。
- 想要快速入门，阅读 [ZeroMQ 指南](#)。

gevent

[gevent](#) 是一款基于协程的Python 网络库，它使用 greenlets 提供在 libev 事件循环之上的高层次异步API。

原文: <http://pythonguidecn.readthedocs.io/zh/latest/scenarios/network.html>

系统管理



Fabric

[Fabric](#) 是一个简化系统管理任务的库。Chef和Puppet倾向于关注管理服务器和系统库，而Fabric更加关注应用级别的任务，比如说部署。

安装Fabric：

```
1. $ pip install fabric
```

下面的代码将会创建我们可以使用的两个任务：[memory_usage](#) 和 [deploy](#)。前者将会在每台机器上输出内存使用情况。后者将会ssh到每台服务器，cd到我们的工程目录，激活虚拟环境，拉取最新的代码库，以及重启应用服务器。

```
1. from fabric.api import cd, env, prefix, run, task
2.
3. env.hosts = ['my_server1', 'my_server2']
4.
5. @task
6. def memory_usage():
7.     run('free -m')
```

```

8.
9. @task
10. def deploy():
11.     with cd('/var/www/project-env/project'):
12.         with prefix('. ./.bin/activate'):
13.             run('git pull')
14.             run('touch app.wsgi')

```

将上述代码保存到文件 `fabfile.py` 中，我们可以这样检查内存的使用：

```

1. $ fab memory_usage
2. [my_server1] Executing task 'memory'
3. [my_server1] run: free -m
4. [my_server1] out:          total    used    free   shared  buffers  cached
5. [my_server1] out: Mem:      6964    1897    5067      0     166     222
6. [my_server1] out: -/+ buffers/cache:  1509    5455
7. [my_server1] out: Swap:      0        0        0
8.
9. [my_server2] Executing task 'memory'
10. [my_server2] run: free -m
11. [my_server2] out:          total    used    free   shared  buffers  cached
12. [my_server2] out: Mem:      1666    902    764      0     180     572
13. [my_server2] out: -/+ buffers/cache:  148    1517
14. [my_server2] out: Swap:     895      1     894

```

and we can deploy with:

```
1. $ fab deploy
```

额外的特性包括并行执行、和远程程序交互、以及主机分组。

[Fabric 文档](#)

Salt

[Salt](#) 是一个开源的基础管理工具。它支持从中心节点（主要的主机）到多个主机（指从机）的远程命令执行。它也支持系统语句，能够使用简单的模板文件配置多台服务器。

Salt支持Python 2.6和2.7，并能通过pip安装：

```
1. $ pip install salt
```

在配置好一台主服务器和任意数量的从机后，我们可以在从机上使用任意的shell命令或者预制的复杂命令的模块。

下面的命令使用ping模块列出所有可用的从机：

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

主机过滤是通过匹配从机id或者使用颗粒系统 (grains system)。颗粒 (grains) 系统使用静态的主机信息，比如操作系统版本或者CPU架构，来为Salt模块提供主机分类内容。

下列命令行使用颗粒系统列举了所有可用的运行CentOS的从机：

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

Salt也提供状态系统。状态能够用来配置从机。

例如，当一个从机接受读取下列状态文件的指令，他将会安装和启动Apache服务器：

```
1. apache:
2.   pkg:
3.     - installed
4.   service:
5.     - running
6.     - enable: True
7.     - require:
8.       - pkg: apache
```

状态文件可以使用YAML、Jinja2模板系统或者纯Python编写。

[Salt 文档](#)

Psutil

[Psutil](#) 是获取不同系统信息（比如CPU、内存、硬盘、网络、用户、进程）的接口。

下面是一个关注一些服务器过载的例子。如果任意一个测试（网络、CPU）失败，它将会发送一封邮件。

```
1. # 获取系统变量的函数:
2. from psutil import cpu_percent, net_io_counters
3. # 休眠函数:
4. from time import sleep
5. # 用于email服务的包:
6. import smtplib
7. import string
8. MAX_NET_USAGE = 400000
9. MAX_ATTACKS = 4
10. attack = 0
11. counter = 0
12. while attack <= MAX_ATTACKS:
13.     sleep(4)
14.     counter = counter + 1
15.     # Check the cpu usage
16.     if cpu_percent(interval = 1) > 70:
17.         attack = attack + 1
18.     # Check the net usage
19.     neti1 = net_io_counters()[1]
```

```

20.     neto1 = net_io_counters()[0]
21.     sleep(1)
22.     neti2 = net_io_counters()[1]
23.     neto2 = net_io_counters()[0]
24.     # Calculate the bytes per second
25.     net = ((neti2+neto2) - (neti1+neto1))/2
26.     if net > MAX_NET_USAGE:
27.         attack = attack + 1
28.     if counter > 25:
29.         attack = 0
30.     counter = 0
31. # 如果attack大于4, 就编写一封十分重要的email
32. TO = "you@your_email.com"
33. FROM = "webmaster@your_domain.com"
34. SUBJECT = "Your domain is out of system resources!"
35. text = "Go and fix your server!"
36. BODY = string.join(("From: %s" %FROM, "To: %s" %TO, "Subject: %s" %SUBJECT, "", text), "\r\n")
37. server = smtplib.SMTP('127.0.0.1')
38. server.sendmail(FROM, [TO], BODY)
39. server.quit()

```

一个类似于基于psutil并广泛扩展的top，并拥有客服端-服务端监控能力的完全终端应用叫做glance。

Ansible

[Ansible](#) 是一个开源系统自动化工具。相比于Puppet或者Chef最大的优点是它不需要客户机上的代理。Playbooks是Ansible的配置、部署和编排语言，它用YAML格式编写，使用Jinja2作为模板。

Ansible支持Python 2.6和2.7，并能使用pip安装：

```
1. $ pip install ansible
```

Ansible需要一个清单文件，来描述主机经过何处。以下是一个主机和playbook的例子，在清单文件中将会ping所有主机。

清单文件示例如下：[hosts.yml](#)

```
1. [server_name]
2. 127.0.0.1
```

playbook示例如下：[ping.yml](#)

```

1. ---
2. - hosts: all
3.
4.   tasks:
5.     - name: ping
6.       action: ping

```

要运行playbook：

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

Ansible playbook在 `hosts.yml` 中将会ping所有的服务器。您也可以选择成组的服务器使用Ansible。了解更多关于Ansible的信息，请阅读 [Ansible Docs](#)。

[Ansible教程](#) 也是一个很棒的且详细的指引来开始熟悉Ansible。

Chef

[Chef](#) 是一个系统的云基础设施自动化框架，它使部署服务器和应用到任何物理、虚拟或者云终端上变得简单。您可以选择进行配置管理，那将主要使用Ruby去编写您的基础设施代码。

Chef客户端运行于组成您的基础设施的每台服务器上，这些客户端定期检查Chef服务器来确保系统是均衡并且处于设想的状态。由于每台服务器拥有它自己的独立的Chef客户端，每个服务器配置自己，这种分布式方法使得Chef成为一个可扩展的自动化平台。

Chef通过使用定制的在cookbook中实现的食谱（配置元素）来工作。Cookbook通常作为基础设施的选择项，作为包存放在Chef服务器中。请阅读 [数字海洋教程系列](#)关于Chef的部分来学习如何创建一个简单的Chef服务器。

要创建一个简单的cookbook，使用 `knife` 命令：

```
1. knife cookbook create cookbook_name
```

[Getting started with Chef](#)对Chef初学者来说是一个好的开始点，许多社区维护着cookbook，可以作为是一个好的参考。要服务自己的基础设施配置需求，请见 [Chef Supermarket](#)。

- [Chef 文档](#)

Puppet

[Puppet](#) 是来自Puppet Labs的IT自动化和配置管理软件，允许系统管理员定义他们的IT基础设施状态，这样就能够提供一种优雅的方式管理他们成群的物理和虚拟机器。

Puppet均可作为开源版和企业版获取到。其模块是小的、可共享的代码单元，用以自动化或定义系统的状态。[Puppet Forge](#) 是一个模块仓库，它由社区编写，面向开源和企业版的Puppet。

Puppet代理安装于其状态需要被监控或者修改的节点上。作为特定服务器的Puppet Master负责组织代理节点。

代理节点发送系统的基本信息到Puppet Master，比如说操作系统、内核、架构、ip地址、主机名等。接着，Puppet Master编译携带有节点生成信息的目录，告知每个节点应如何配置，并发送给代理。代理便会执行前述目录中的变化，并向Puppet Master发送回一份报告。

Facter是一个有趣的工具，它用来传递Puppet获取到的基本系统信息。这些信息可以在编写Puppet模块的时候作为变量来引用。

```
1. $ facter kernel
2. Linux
```

```
1. $ facter operatingsystem
2. Ubuntu
```

在Puppet中编写模块十分直截了当。Puppet清单（manifest）组成了Puppet模块。Puppet清单以扩展名 `.pp` 结尾。下面是一个Puppet中 ‘Hello World’ 的例子。

```
1. notify { 'This message is getting logged into the agent node':
2.
3.   #As nothing is specified in the body the resource title
4.   #the notification message by default.
5. }
```

这里是另一个基于系统的逻辑的例子。注意操纵系统信息是如何作为变量使用的，变量前加了前缀符号 `$`。类似的，其他信息比如说主机名就能用 `$hostname` 来引用。

```
1. notify{ 'Mac Warning':
2.   message => $operatingsystem ? {
3.     'Darwin' => 'This seems to be a Mac.',
4.     default  => 'I am a PC.',
5.   },
6. }
```

Puppet有多种资源类型，需要时可以使用包-文件-服务（package-file-service）范式来承担配置管理的主要任务。下面的Puppet代码确保了系统中安装了OpenSSH-Server包，并且在每次sshd配置文件改变时重启sshd服务。

```
1. package { 'openssh-server':
2.   ensure => installed,
3. }
4.
5. file { '/etc/ssh/sshd_config':
6.   source  => 'puppet:///modules/sshd/sshd_config',
7.   owner   => 'root',
8.   group   => 'root',
9.   mode    => '640',
10.  notify   => Service['sshd'], # sshd will restart
11.          # whenever you edit this
12.          # file
13.  require  => Package['openssh-server'],
14.
15. }
16.
17. service { 'sshd':
18.   ensure     => running,
19.   enable     => true,
20.   hasstatus  => true,
21.   hasrestart=> true,
```

22. }

了解更多信息，参考 [Puppet Labs 文档](#)。

Blueprint

待处理

[Write about Blueprint](#)

Buildout

[Buildout](#) 是一个开源软件构件工具。Buildout由Python编写。它实现了配置和构建脚本分离的原则。Buildout主要用于下载和设置正在开发或部署软件的Python egg格式的依赖。在任何环境中构建任务的指南（recipe，原意为“食谱”，引申为“指南”）能被创建，许多早已可用。

Shinken

[Shinken](#) 是一个使用Python编写的现代化的兼容Nagios的监控框架。其主要目标是为用户的设计成可扩展到大型环境的监控系统提供灵活的框架。

Shinken与Nagios配置标准和插件向后兼容。它适用于任何支持Python的操作系统和架构，包括Windows、GNU/Linux和FreeBSD。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/admin.html>

持续集成



注解

关于编写测试更多的建议，参阅 [测试您的代码](#)。

为什么？

Martin Fowler 和 Kent Beck 首次提出 [Continuous Integration](#)（简称：CI），将之描述为：

持续集成是一种软件开发实践：许多团队频繁地集成他们的工作，每位成员通常进行日常集成，进而每天会有多种集成。每个集成会由自动的构建（包括测试）来尽可能快地检测错误。许多团队发现这种方法可以显著的减少集成问题并且可以使团队的开发更加快捷。

Jenkins

[Jenkins CI](#) 可扩展的持续集成引擎。 使用它吧！

Buildbot

[Buildbot](#) 是一个检查代码变化的自动化编译/测试的Python系统。

Tox

[Tox](#) 是一款为Python软件提供打包、测试和开发的自动化工具，基于命令行或CI服务器。它是一个通用的虚拟环境管理和测试的命令行工具，提供如下特性：

- 检查包在不同的Python版本和解释器下安装正确
- 在每个环境中运行您的测试、配置测试工具的选择
- 作为前端持续集成服务器，减少样板文件，合并了CI和基于shell的测试。

Travis-CI

[Travis-CI](#) 是一个分布式CI服务器，免费为开源项目构建测试。它提供多个worker运行Python测试，并能和GitHub无缝集成。您甚至可以用它对您的Pull Requests评论是否构建这个特定的变更集。[Travis-ci](#)是一个很好的、简单的方式去了解持续集成。

作为开始，将 `.travis.yml` 文件加入到您的仓库中，内容如下：

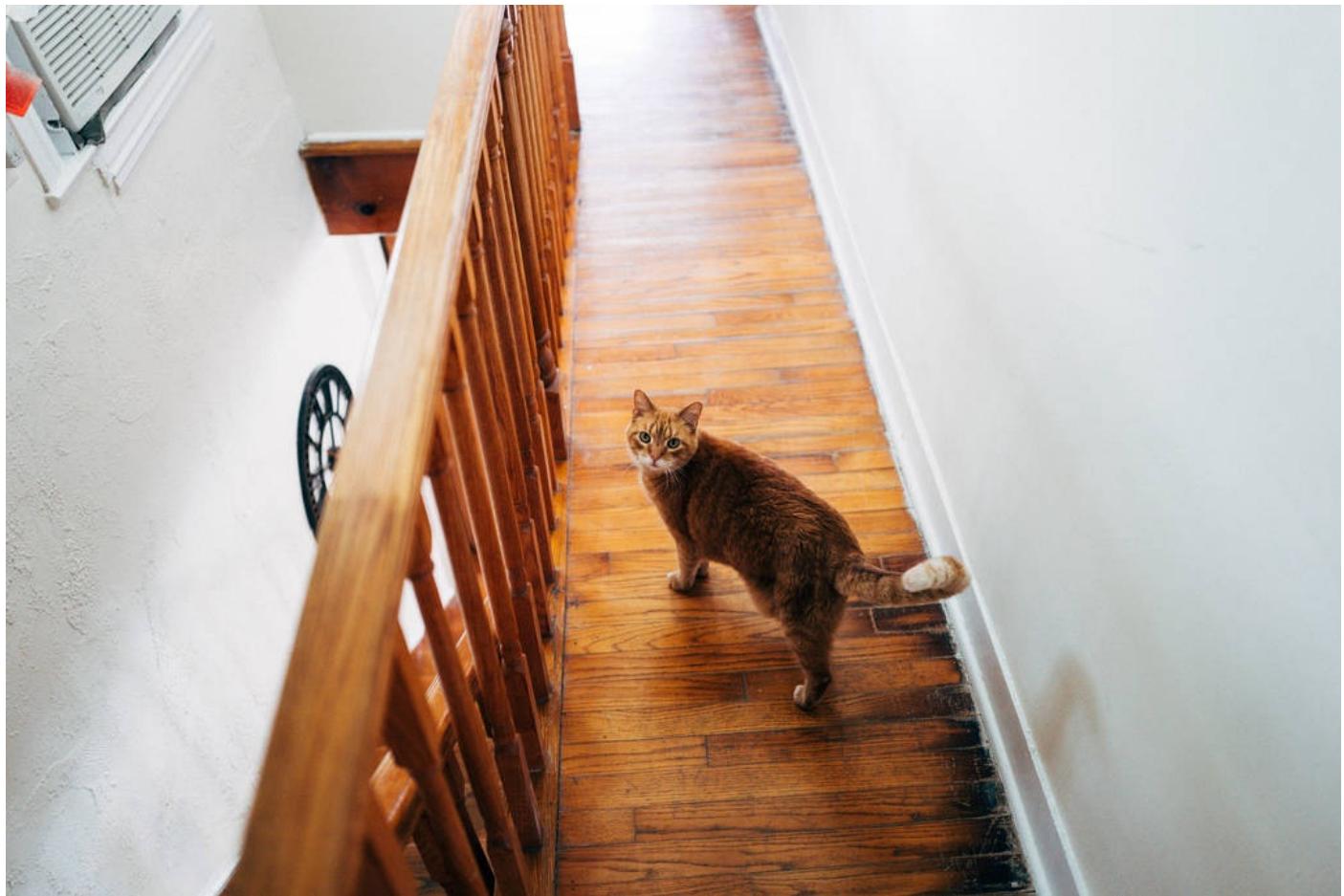
```
1. language: python
2. python:
3.   - "2.6"
4.   - "2.7"
5.   - "3.2"
6.   - "3.3"
7. # command to install dependencies
8. script: python tests/test_all_of_the_units.py
9. branches:
10. only:
11.   - master
```

这将会使您的工程在罗列的Python版本中，用给定的脚本进行测试，而且只会构建主干分支。有许多可供开启的选项，包括通知、步骤前后等。[Travis-ci docs](#)详尽地解释了所有这些操作。

为了激活您的工程的测试，去 [travis-ci 网站](#) 登录您的GitHub账号。然后在您的profile设置中激活您的工程。现在，每一次push到GitHub上的提交将会运行您的工程中的测试。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/ci.html>

速度



CPython作为最流行的Python环境，对于CPU密集型任务（CPU bound tasks）较慢，而 PyPy 则较快。

使用稍作改动的 [David Beazley](#) 的 CPU密集测试代码（增加了循环进行多轮测试），您可以看到CPython与PyPy之间的执行差距。

```
1. # PyPy
2. $ ./pypy -V
3. Python 2.7.1 (7773f8fc4223, Nov 18 2011, 18:47:10)
4. [PyPy 1.7.0 with GCC 4.4.3]
5. $ ./pypy measure2.py
6. 0.0683999061584
7. 0.0483210086823
8. 0.0388588905334
9. 0.0440690517426
10. 0.0695300102234
```

```
1. # CPython
2. $ ./python -V
3. Python 2.7.1
4. $ ./python measure2.py
5. 1.06774401665
6. 1.45412397385
```

```

7. 1.51485204697
8. 1.54693889618
9. 1.60109114647

```

Context

The GIL

[GIL](#) (全局解释器锁)是Python支持多线程并行操作的方式。Python的内存管理不是线程安全的，所以GIL被创造出来避免多线程同时运行同一个Python代码。

David Beazley 有一个关于GIL如何工作的 [指导](#)。他也讨论了 Python3.2中的 [新GIL](#)他的结论是为了最大化一个Python程序的性能，应该对GIL工作方式有一个深刻的理解 ——它如何影响您的特定程序，您拥有多少核，以及您程序瓶颈在哪。

C 扩展

GIL

当写一个C扩展时必须 [特别关注](#) 在解释器中注册您的线程。

C 扩展

Cython

[Cython](#) 是Python语言的一个超集，对其您可以为Python写C或C++模块。Cython也使得您可以从已编译的C库中调用函数。使用Cython让您得以发挥Python的变量与操作的强类型优势。

这是一个cython中的强类型例子。

```

1. def primes(int kmax):
2.     """有一些Cython附加关键字的素数计算"""
3.
4.     cdef int n, k, i
5.     cdef int p[1000]
6.     result = []
7.     if kmax > 1000:
8.         kmax = 1000
9.     k = 0
10.    n = 2
11.    while k < kmax:
12.        i = 0
13.        while i < k and n % p[i] != 0:
14.            i = i + 1
15.        if i == k:
16.            p[k] = n

```

```

17.         k = k + 1
18.         result.append(n)
19.         n = n + 1
20.     return result

```

将这个有一些附加关键字的寻找素数算法实现与下面这个纯Python实现比较：

```

1. def primes(kmax):
2. """标准Python语法下的素数计算"""
3.
4.     p = range(1000)
5.     result = []
6.     if kmax > 1000:
7.         kmax = 1000
8.     k = 0
9.     n = 2
10.    while k < kmax:
11.        i = 0
12.        while i < k and n % p[i] != 0:
13.            i = i + 1
14.        if i == k:
15.            p[k] = n
16.            k = k + 1
17.            result.append(n)
18.        n = n + 1
19.    return result

```

注意，在Cython版本，在创建一个Python列表时，您声明了会被编译为C类型的整型和整型数组。

```

1. def primes(int kmax):
2. """有一些Cython附加关键字的素数计算 """
3.
4.     cdef int n, k, i
5.     cdef int p[1000]
6.     result = []

```

```

1. def primes(kmax):
2. """标准Python语法下的素数计算"""
3.
4.     p = range(1000)
5.     result = []

```

有什么差别呢？在上面的Cython版本中，您可以看到变量类型与整型数组像标准C一样被声明。作为例子，第三行的 `cdef int n, k, i` 这个附加类型声明（整型）使得Cython编译器得以产生比第二个版本更有效率的C代码。标准Python代码以 `*.py` 格式保存，而 Cython 以 `*.pyx` 格式保存。

速度上有什么差异呢？看看这个！

```

1. import time
2. #启动pyx编译器

```

```

3. import pyximport
4. pyximport.install()
5. #Cython的素数算法实现
6. import primesCy
7. #Python的素数算法实现
8. import primes
9.

10. print "Cython:"
11. t1= time.time()
12. print primesCy.primes(500)
13. t2= time.time()
14. print "Cython time: %s" %(t2-t1)
15. print ""
16. print "Python"
17. t1= time.time()
18. print primes.primes(500)
19. t2= time.time()
20. print "Python time: %s" %(t2-t1)

```

这两行代码需要一些说明：

```

1. import pyximport
2. pyximport.install()

```

`pyximport` 使得您可以导入 `*.pyx` 文件，（像 `primesCy.pyx` 这样的）。`pyximport.install()` 命令使 Python 解释器可以打开 Cython 编译器直接编译出 `*.so` 格式的 C 库。Cython 之后可以导入这个库到您的 Python 代码中，简便而有效。使用 `time.time()` 函数您可以比较两个不同的在查找 500 个素数的调用长的时间消耗差异。在一个标准笔记本中（双核 AMD E-450 1.6GHz），测量值是这样的：

```

1. Cython time: 0.0054 seconds
2.
3. Python time: 0.0566 seconds

```

而这个是嵌入的 `ARM beaglebone` 机的输出结果：

```

1. Cython time: 0.0196 seconds
2.
3. Python time: 0.3302 seconds

```

Pyrex

Shedskin?

Concurrency

Concurrent.futures

`concurrent.futures` 模块是标准库中的一个模块，它提供了一个“用于异步调用的高级接口”。它抽象了许多关于使用多个线程或进程并发的更复杂的细节，并允许用户专注于完成手头的任务。

`concurrent.futures` 模块提供了两个主要的类，即 `ThreadPoolExecutor` 和 `ProcessPoolExecutor`。
`ThreadPoolExecutor` 将创建一个用户可以提交作业的工作线程池。当下一个工作线程可用时，这些作业将在另一个线程中执行。

`ProcessPoolExecutor` 以相同的方式工作，它使用多进程而不是多线程作为工作池。这就可以避开GIL的问题，但是由于传递参数给工作进程的工作原理，只有可序列化的对象可以执行并返回。

由于GIL的工作原理，一个很好的经验法则是当执行涉及很多阻塞（如通过网络发出请求）的任务时使用 `ThreadPoolExecutor`，而对高计算开销的任务使用 `ProcessPoolExecutor` 执行器。

使用两个执行器并行执行有两个主要方法。一个是使用 `map(func, iterables)` 方法。这个函数除了能并行执行一切，它几乎和内置的 `map()` 函数一模一样：

```

1. from concurrent.futures import ThreadPoolExecutor
2. import requests
3.
4. def get_webpage(url):
5.     page = requests.get(url)
6.     return page
7.
8. pool = ThreadPoolExecutor(max_workers=5)
9.
10. my_urls = ['http://google.com/']*10 # Create a list of urls
11.
12. for page in pool.map(get_webpage, my_urls):
13.     # 处理结果
14.     print(page.text)

```

为了进一步的控制，`submit(func, args, *kwargs)` 方法将调度一个可执行的调用（如 `func(args, *kwargs)`），并返回一个代表可调用的执行的 `Future` 对象。

`Future` 对象提供了可用于检查计划可调用进程的各种方法。这些包括：

`cancel()` 尝试取消调用。`cancelled()` 如果调用被成功取消，返回`True`。`running()` 如果当前正在执行调用而且没被取消，则返回`True`。`done()` 如果调用被成功取消或完成运行，返回`True`。`result()` 返回调用返回的值。请注意，此调用将阻塞到默认情况下调度的可调用对象的返回。`exception()` 返回调用抛出的异常。如果没有抛出异常，将返回 `None`。请注意，这和 `result()` 一样会阻塞。`add_done_callback(fn)` 添加回调函数函数，在所调用的可调用对象执行返回时执行（如 `_fn(future)`）。预定可回拨。

```

1. from concurrent.futures import ProcessPoolExecutor, as_completed
2.
3. def is_prime(n):
4.     if n % 2 == 0:
5.         return n, False
6.
7.     sqrt_n = int(n**0.5)
8.     for i in range(3, sqrt_n + 1, 2):
9.         if n % i == 0:

```

```

10.         return n, False
11.     return n, True
12.
13. PRIMES = [
14.     112272535095293,
15.     112582705942171,
16.     112272535095293,
17.     115280095190773,
18.     115797848077099,
19.     1099726899285419]
20.
21. futures = []
22. with ProcessPoolExecutor(max_workers=4) as pool:
23.     # Schedule the ProcessPoolExecutor to check if a number is prime
24.     # and add the returned Future to our list of futures
25.     for p in PRIMES:
26.         fut = pool.submit(is_prime, p)
27.         futures.append(fut)
28.
29. # As the jobs are completed, print out the results
30. for number, result in as_completed(futures):
31.     if result:
32.         print("{} is prime".format(number))
33.     else:
34.         print("{} is not prime".format(number))

```

`concurrent.futures` 模块包含两个帮助函数来处理Futures。`as_completed(futures)` 函数返回futures列表的迭代器，在futures结束时yield。

而 `wait(futures)` 函数则简单地阻塞，直到列表中所有的futures完成。

有关使用 `concurrent.futures` 模块的更多信息，请参阅官方文档。

Threading

标准库带有一个 `threading` 模块，允许用户手动处理多个线程。

在另一个线程中运行一个函数就如传递一个可调用对象以及它的参数到 `Thread` 的构造函数中，然后调用 `start()` 一样简单：

```

1. from threading import Thread
2. import requests
3.
4. def get_webpage(url):
5.     page = requests.get(url)
6.     return page
7.
8. some_thread = Thread(get_webpage, 'http://google.com/')
9. some_thread.start()

```

调用 `join()` 来等待线程终止：

```
1. some_thread.join()
```

调用 `join()` 后，检查线程是否仍然存在（因为 `join` 调用超时）总是一个好主意：

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

由于多个线程可以访问相同的内存部分，有时可能会出现两个或多个线程尝试同时写入同一资源的情况，或者输出取决于某些事件的顺序或时序。这被称为 **数据竞争** 或 **竞争条件**。当这种情况发生时，输出将会出现乱码，或者可能会遇到难以调试的问题。[stackoverflow post](#) 是个很好的例子。

可以避免的方法是每个线程在写入共享资源之前获取 **Lock**。锁可以通过环境上下文协议（`with` 语句）或直接使用 `acquire()` 和 `release()` 来获取和释放。以下是一个（颇有争议的）例子：

```
1. from threading import Lock, Thread
2.
3. file_lock = Lock()
4.
5. def log(msg):
6.     with file_lock:
7.         open('website_changes.log', 'w') as f:
8.             f.write(changes)
9.
10. def monitor_website(some_website):
11.     """
12.     Monitor a website and then if there are any changes,
13.     log them to disk.
14.     """
15.     while True:
16.         changes = check_for_changes(some_website)
17.         if changes:
18.             log(changes)
19.
20. websites = ['http://google.com/', ...]
21. for website in websites:
22.     t = Thread(monitor_website, website)
23.     t.start()
```

在这里，我们有一堆线程检查站点列表中的更改，每当有任何更改时，它们尝试通过调用 `log(changes)` 将这些更改写入文件。当调用 `log()` 时，它在 `with file_lock:` 处等待获取锁。这样可以确保在任何时候只有一个线程正在写入文件。

Spawning Processes

Multiprocessing

原文: <http://pythonguidecn.readthedocs.io/zh/latest/scenarios/speed.html>

科学应用



背景

Python被经常使用在高性能科学应用中。它之所以在学术和科研项目中得到如此广泛的应用是因为它容易编写而且执行效果很好。

由于科学计算对高性能的要求，Python中相关操作经常借用外部库，通常是以更快的语言（如C，或者FORTRAN来进行矩阵操作）写的。其主要的库有 [Numpy](#), [Scipy](#) 以及 [Matplotlib](#)。关于这些库的细节超出了本指南的范围。不过，关于Python的科学计算生态的综合介绍可以在这里找到[Python Scientific Lecture Notes](#)。

工具

IPython

[IPython](#) 是一个加强版Python解释器，它提供了科学工作者感兴趣的特性。其中，*inline mode* 允许将图像绘制到终端中（基于Qt）。进一步的，*notebook* 模式支持文学化编程（literate programming，译者注：作者这里可能是指其富文本性不是那个编程范式）与可重现性（reproducible，译者注：作者可能是指每段程序可以单独重新计算的特性），它产生了一个基于web的python 笔记本。这个笔记本允许您保存一些代码块，伴随着它们的计算结果以及增强的注释（HTML, LaTex, Markdown）。这个笔记本可以被共享并以各种文件格式导出。

库

NumPy

[NumPy](#) 是一个用C和FORTRAN写的底层库，它提供一些高阶数学函数。NumPy通过多维数组和操作这些数组的函数巧妙地解决了Python运行算法较慢的问题。任何算法只要被写成数组中的函数，就可以运行得很快。

NumPy是SciPy项目中的一部分，它被发布为一个独立的库，这样对于只需基本功能的人来说，就不用安装SciPy的其余部分。

NumPy兼容Python 2.4-2.7.2以及3.1+。

Numba

[Numba](#) 是一个针对NumPy的Python编译器（即时编译器，JIT）它通过特殊的装饰器，将标注过的Python（以及NumPy）代码编译到LLVM（Low Level Virtual Machine，底层虚拟机）中。简单地说，Python使用一种机制，用LLVM将Python代码编译为能够在运行时执行的本地代码。

SciPy

[SciPy](#) 是基于NumPy并提供了更多的数学函数的库。SciPy使用NumPy数组作为基本数据结构，并提供完成各种常见科学编程任务的模块，包括线性代数，积分（微积分），常微分方程求解以及信号过程。

Matplotlib

[Matplotlib](#) 是一个可以灵活绘图的库，它能够创建2D、3D交互式图形，并能保存成具有稿件质量（manuscript-quality）的图表。其API很像 [MATLAB](#)，这使得MATLAB用户很容易转移到Python。在 [matplotlib gallery](#) 中可以找到很多例子以及实现它们的源代码（可以在此基础上再创造）。

Pandas

[Pandas](#) 是一个基于NumPy的数据处理库，它提供了许多有用的函数能轻松地对数据进行访问、索引、合并以及归类。其主要数据结构（DataFrame）与R统计学包十分相近；也就是，使用名称索引的异构数据（heterogeneous data）表、时间序列操作以及对数据的自动对准（auto-alignment）。

Rpy2

[Rpy2](#) 是一个对R统计学包的Python绑定，它能够让Python执行R函数，并在两个环境中交换数据。Rpy2是对 [Rpy](#) 绑定的面向对象实现。

PsychoPy

[PsychoPy](#) 是面向认知科学家的库，它允许创建认知心理学和神经科学实验（译者注：指的是那种您坐在电脑前，给您一个刺激测您反应的实验，基本上就是个UI）。这个库能够处理刺激表示、实验设计脚本以及数据收集。

资源

安装这些科学计算Python包可能会有些麻烦，因为它们中很多是用Python的C扩展实现的，这就意味着需要编译。这一节列举了各种科学计算Python发行版，它们提供了预编译编译且易于安装的科学计算Python包。

Python扩展包的非官方Windows二进制文件（库）

很多人在Windows平台上做科学计算，然而众所周知的是，其中很多科学计算包在该平台上难以构建和安装。不过，[Christoph Gohlke](#) 将一系列有用的Python包编译成了Windows的二进制文件，其数量还在不断增长。如果您在Windows上工作，您也许想要看看。

Anaconda

[Continuum Analytics](#) 提供了[Anaconda Python Distribution](#)，它拥有所有常见的Python科学包，也包括与数据分析和大数据相关的包。Anaconda是免费的而Continuum销售一些专有的额外组件。学术研究者可以获取这些组件的免费许可。

Canopy

[Canopy](#) 是另一个Python科学发布版，由[Enthought](#) 提供。其受限制的 'Canopy Express' 版本是免费提供的，但是Enthought负责完整版。学术研究者可以获取到免费许可。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/scientific.html>

图像处理



多数图像处理与操作技术可以被两个库有效完成，它们是Python Imaging Library (PIL)与OpenSource Computer Vision (OpenCV)。

下面是这两个库的简略介绍。

Python 图形库

Python Imaging Library，或者叫PIL，简略来说，是Python图像操作的核心库。不幸的是，它的开发陷入了停滞，最后一次更新是2009年。对您而言幸运的是，存在一个活跃的PIL开发分支，叫做 Pillow它很容易安装，运行在各个操作系统上，而且支持Python3。

安装

在安装Pillow之前，您应该先安装Pillow的前置部分。针对您的平台对此的特别指导可以在此找到[Pillow installation instructions](#).

完成之后，直接执行：

```
1. $ pip install Pillow
```

例子

```

1. from PIL import Image, ImageFilter
2. #读取图像
3. im = Image.open( 'image.jpg' )
4. #显示图像
5. im.show()
6.
7. #过滤图像
8. im_sharp = im.filter( ImageFilter.SHARPEN )
9. #保存过滤过的图像到文件中
10. im_sharp.save( 'image_sharpened.jpg', 'JPEG' )
11.
12. #分解图像到三个RGB不同的通道（band）中。
13. r,g,b = im_sharp.split()
14.
15. #显示被插入到图像中的EXIF标记
16. exif_data = im._getexif()
17. exif_data

```

这里有一些Pillow库的例子：[Pillow 教程](#)。

开源计算机视觉（OpenCv）

OpenSource Computer Vision，其更广为人知的名字是OpenCv，是一个在图像操作与处理上比PIL更先进的库。它可以在很多语言上被执行并被广泛使用。

安装

在Python中，使用OpenCV进行图像处理是通过使用 [cv2](#) 与 [NumPy](#) 模块进行的。[OpenCV 安装指南](#)可以指导您如何为您自己的项目进行配置。

NumPy可以从Python Package Index（PyPI）中下载：

```
1. $ pip install numpy
```

例子

```

1. import cv2
2. #读取图像
3. img = cv2.imread('testimg.jpg')
4. #显示图像
5. cv2.imshow('image',img)
6. cv2.waitKey(0)
7. cv2.destroyAllWindows()
8.
9. #Applying Grayscale filter to image 作用Grayscale（灰度）过滤器到图像上
10. gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

```

```
11.  
12. #保存过滤过的图像到新文件中  
13. cv2.imwrite('graytest.jpg',gray)
```

更多的OpenCV在Python运行例子在这里可以找到: [collection oftutorials.](#)

原文: <http://pythonguidecn.readthedocs.io/zh/latest/scenarios/imaging.html>

数据序列化



什么是数据序列化？

数据序列化是将结构化数据转换成允许以共享或存储的格式，可恢复其原始结构的概念。在某些情况下，数据序列化的第二个目的是将要序列化数据的大小最小化，从而使磁盘空间或带宽要求最小化。

Pickle

Python原生的数据序列化模块称为 `Pickle`。

例子如下：

```
1. import pickle
2.
3. #Here's an example dict
4. grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }
5.
6. #Use dumps to convert the object to a serialized string
7. serial_grades = pickle.dumps( grades )
8.
9. #Use loads to de-serialize an object
```

```
10. received_grades = pickle.loads( serial_grades )
```

Protobuf

如果您正在寻找支持多种语言的序列化模块，那么Google的 **Protobuf** 库就是一个选择。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/serialization.html>

XML解析



untangle

`untangle` 库可以将XML文档映射为一个Python对象，该对象于其结构中包含了原文档的节点与属性信息。

作为例子，一个像这样的XML文件：

```
1. <?xml version="1.0"?>
2. <root>
3.   <child name="child1">
4. </root>
```

可以被这样载入：

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

然后您可以像这样获取child元素名称：

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

untangle也支持从字符串或URL中载入XML。

xmltodict

`xmltodict` 是另一个简易的库，它致力于将XML变得像JSON。

对于一个像这样的XML文件：

```
1. <mydocument has="an attribute">
2.   <and>
3.     <many>elements</many>
4.     <many>more elements</many>
5.   </and>
6.   <plus a="complex">
7.     element as well
8.   </plus>
9. </mydocument>
```

可以装载进一个Python字典里，像这样：

```
1. import xmltodict
2.
3. with open('path/to/file.xml') as fd:
4.     doc = xmltodict.parse(fd.read())
```

您可以访问元素，属性以及值，像这样：

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

`xmltodict` 也有`unparse`函数让您可以转回XML。该函数有一个`streaming`模式适合用来处理不能放入内存的文件，它还支持命名空间。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/xml.html>

JSON



`json` 库可以自字符串或文件中解析JSON。该库解析JSON后将其转为Python字典或者列表。它也可以转换Python字典或列表为JSON字符串。

解析JSON

创建下面包含JSON数据的字符串

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

它可以被这样解析：

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

然后它可以像一个常规的字典那样使用：

```
1. print(parsed_json['first_name'])  
2. "Guido"
```

您可以把下面这个对象转为JSON：

```
1. d = {  
2.     'first_name': 'Guido',  
3.     'second_name': 'Rossum',  
4.     'titles': ['BDFL', 'Developer'],  
5. }  
6.  
7. print(json.dumps(d))  
8. '{"first_name": "Guido", "last_name": "Rossum", "titles": ["BDFL", "Developer"]}'
```

simplejson

JSON库是Python2.6版中加入的。如果您使用更早版本的Python，可以通过PyPI获取 [simplejson](#) 库。

simplejson类似json标准库，它使得使用老版本Python的开发者们可以使用json库中的最新特性。

如果json库不可用，您可以将simplejson取别名为json来使用：

```
1. import simplejson as json
```

在将simplejson当成json导入后，上面的例子会像您在使用标准json库一样正常运行。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/json.html>

密码学



Cryptography

[Cryptography](#) 是一个开发活跃的库，它提供了加密方法（recipes）和基元（primitives），支持Python 2.6-2.7、Python 3.3+ 和 PyPy。

Cryptography 分为两个层，方法（recipes）层和危险底层（hazardous materials，简称hazmat）。方法层提供用于适当的对称加密，hazmat层提供底层的加密基元。

安装

```
1. $ pip install cryptography
```

例子

示例代码使用了高层的对称加密方法：

```
1. from cryptography.fernet import Fernet  
2. key = Fernet.generate_key()  
3. cipher_suite = Fernet(key)
```

```

4. cipher_text = cipher_suite.encrypt(b"A really secret message. Not for prying eyes.")
5. plain_text = cipher_suite.decrypt(cipher_text)

```

GPGME bindings

[GPGME Python bindings](#) 提供Pythonic的方式访问 [GPG Made Easy](#)，这是整个GNU Privacy Guard项目套件，包括GPG、libgcrypt和gpgsm（S/MIME引擎），的C API。它支持Python 2.6、2.7、3.4及以上版本。取决于Python的SWIG C接口以及GnuPG软件和库。

这里有更全面的GPGME Python Bindings HOWTO的 [源码版](#) 和 [HTML版](#)。还提供了Python 3版本的HOWTO示例脚本的源代码，并且可以在 [这里](#) 访问。

其在与GnuPG其余项目的相同条款（GPLv2和LGPLv2.1，均带有“或更高版本”）下可用。

安装

如果配置脚本定位到了所支持的python版本（配置时位于\$PATH中），那么在编译GPGME时会默认包含它。

例子

```

1. import gpg
2.
3. # Encryption to public key specified in rkey.
4. a_key = input("Enter the fingerprint or key ID to encrypt to: ")
5. filename = input("Enter the filename to encrypt: ")
6. with open(filename, "rb") as afile:
7.     text = afile.read()
8. c = gpg.core.Context(armor=True)
9. rkey = list(c.keylist(pattern=a_key, secret=False))
10. ciphertext, result, sign_result = c.encrypt(text, recipients=rkey,
11.                                              always_trust=True,
12.                                              add_encrypt_to=True)
13. with open("{0}.asc".format(filename), "wb") as bfile:
14.     bfile.write(ciphertext)
15. # Decryption with corresponding secret key
16. # invokes gpg-agent and pinentry.
17. with open("{0}.asc".format(filename), "rb") as cfile:
18.     plaintext, result, verify_result = gpg.Context().decrypt(cfile)
19. with open("new-{0}".format(filename), "wb") as dfile:
20.     dfile.write(plaintext)
21. # Matching the data.
22. # Also running a diff on filename and the new filename should match.
23. if text == plaintext:
24.     print("Hang on ... did you say *all* of GnuPG? Yep.")
25. else:
26.     pass

```

PyCrypto

PyCrypto 是另一个密码库，它提供安全的哈希函数和各种加密算法，支持Python 2.1到3.3。

安装

```
1. $ pip install pycrypto
```

例子

```
1. from Crypto.Cipher import AES
2. # Encryption
3. encryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
4. cipher_text = encryption_suite.encrypt("A really secret message. Not for prying eyes.")
5.
6. # Decryption
7. decryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
8. plain_text = decryption_suite.decrypt(cipher_text)
```

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/crypto.html>

机器学习



Python拥有大量的数据分析、统计和机器学习库，使其成为许多数据科学家的首选语言。

一些广泛使用的机器学习和其他数据科学应用程序包被列在下面：

Scipy 栈 (Scipy stack)

Scipy 栈由数据科学所使用的一组核心帮助包组成，用于统计分析和数据可视化。由于其庞大的功能和易用性，scipy栈被认为是大多数数据科学应用的必备条件。

该栈包含以下包（提供文档链接）：

- [NumPy](#)
- [SciPy library](#)
- [Matplotlib](#)
- [IPython](#)
- [pandas](#)
- [Sympy](#)
- [nose](#)

该栈还附带了Python，但已被排除在上面的列表中。

安装

要安装完整的栈或单个包，您可以参考 [这里](#)给出的说明。

注意：Anaconda 是高推荐的，因为它可以无缝地安装和维护数据科学包。

scikit-learn

Scikit是一个用于Python的免费开源机器学习库。 它提供了现成的功能来实现诸如线性回归、分类器、SVM、k-均值和神经网络等多种算法。它还有一些可直接用于训练和测试的样本数据集。

由于其速度、鲁棒性和易用性，它是许多机器学习应用程序中使用最广泛的库之一。

安装

通过 PyPI：

```
1. pip install -U scikit-learn
```

通过 conda：

```
1. conda install scikit-learn
```

scikit-learn 也随Anaconda发行（如上所述）。 有关更多安装说明，请参阅[此链接](#)。

例子

本例中，我们在 [Iris 数据集](#)上训练一个简单的分类器，它与scikit-learn捆绑在一起。

数据集具有花的四个特征：萼片长度，萼片宽度，花瓣长度和花瓣宽度，并将它们分为三个花种（标签）：setosa、versicolor或virginica。 标签已经被表示为数据集中的数字：0 (setosa)，1 (versicolor) 和 2 (virginica)。

我们清洗Iris数据集，并将其分为独立的训练和测试集：保留最后10个数据点进行测试，剩余的进行训练。然后我们在训练集训练分类器，并对测试集进行预测。

```
1. from sklearn.datasets import load_iris
2. from sklearn import tree
3. from sklearn.metrics import accuracy_score
4. import numpy as np
5.
6. #loading the iris dataset
7. iris = load_iris()
8.
9. x = iris.data #array of the data
10. y = iris.target #array of labels (i.e answers) of each data entry
11.
12. #getting label names i.e the three flower species
13. y_names = iris.target_names
14.
```

```
15. #taking random indices to split the dataset into train and test
16. test_ids = np.random.permutation(len(x))
17.
18. #splitting data and labels into train and test
19. #keeping last 10 entries for testing, rest for training
20.
21. x_train = x[test_ids[:-10]]
22. x_test = x[test_ids[-10:]]
23.
24. y_train = y[test_ids[:-10]]
25. y_test = y[test_ids[-10:]]
26.
27. #classifying using decision tree
28. clf = tree.DecisionTreeClassifier()
29.
30. #training (fitting) the classifier with the training set
31. clf.fit(x_train, y_train)
32.
33. #predictions on the test dataset
34. pred = clf.predict(x_test)
35.
36. print pred #predicted labels i.e flower species
37. print y_test #actual labels
38. print (accuracy_score(pred, y_test))*100 #prediction accuracy
```

由于我们在每次迭代中随机分割和分类训练，所以准确性可能会有所不同。运行上面的代码得到：

```
1. [0 1 1 1 0 2 0 2 2 2]
2. [0 1 1 1 0 2 0 2 2 2]
3. 100.0
```

第一行包含由我们的分类器预测的测试数据的标签（即花种），第二行包含数据集中给出的实际花种。我们这次准确率达到100%。

关于scikit-learn的更多内容可以在 [文档](#)中阅读。

原文: <http://pythonguidecn.readthedocs.io/zh/latest/scenarios/ml.html>

与C/C++库交互



C语言外部函数接口(CFFI)

[CFFI](#) 通过CPython和PyPy给出了和C语言交互的简单使用机制。它支持两种模式：一种是内联的ABI兼容模式(示例如下)，它允许您动态加载和运行可执行模块的函数(本质上与LoadLibrary和dlopen拥有相同的功能)；另一种为API模式，它允许您构建C语言扩展模块。

ABI 交互

```
1. from cffi import FFI
2. ffi = FFI()
3. ffi.cdef("size_t strlen(const char*);")
4. clib = ffi.dlopen(None)
5. length = clib.strlen("String to be evaluated.")
6. # prints: 23
7. print("{}\n".format(length))
```

ctypes

`ctypes` 是CPython中与C/C++交互的事实上的库。它不仅能完全访问大多数主流操作系统(比如: Windows上的Kernel32, *nix上的libc)的纯C接口，并且支持对动态库的加载和交互，如DLL和运行时共享对象。它同时涵盖许多可和系统API交互的类型，并允许您以相对简单的方式定义自己的复杂类型，如`struct`和`union`，并在需要时允许您作出如填充、对齐这样的修改。对它的使用可能稍显复杂，但与 `struct` 模块配合使用，可通过纯C(++)方法让您从根本上控制您的数据类型转换成更有用的东西。

Struct Equivalents

MyStruct.h

```
1. struct my_struct {
2.     int a;
3.     int b;
4. };
```

MyStruct.py

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

SWIG

`SWIG` 并不仅仅应用于Python(它支持多种脚本语言)，它是生成解释性语言和C/C++头文件绑定的工具。它极易使用：使用者只需简单的定义接口文件(详见相关指南和文档)，包含必要的C/C++头文件，并对它们运行生成工具。但它也有其局限性，目前，它与C++部分新特性间仍存在问题，而模板重码的工作多少有些冗繁。只需少量的工作，它便能提供诸多作用，并展现Python的许多特性。同时，您可以简单的扩展SWIG生成的绑定(在接口文件中)来重载操作符和内建函数，也可以有效的重新转换C++异常，使其可被Python所捕获。

例子：Overloading `repr`

MyClass.h

```
1. #include <string>
2. class MyClass {
3. private:
4.     std::string name;
5. public:
6.     std::string getName();
7. };
```

myclass.i

```
1. %include "string.i"
2.
3. %module myclass
4. %{
```

```
5. #include <string>
6. #include "MyClass.h"
7. %}
8.
9. %extend MyClass {
10.     std::string __repr__()
11.     {
12.         return $self->getName();
13.     }
14. }
15.
16. %include "MyClass.h"
```

Boost.Python

Boost.Python需要一些手动工作来展现C++对象的功能，但它可提供SWIG拥有的所有特性。同时，它可提供在C++中访问Python对象的封装，也可提取SWIG封装的对象，甚至可在C++代码中嵌入部分Python。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/scenarios/clibs.html>

- 打包您的代码
- 冻结(freezing)您的代码

打包您的代码



打包就是把您的代码分享给其他开发者。作为例子，共享一个库给其他开发者用在它们的应用中，或者作为一个开发工具，比如 '`py.test`'。

这种分发方式的一个优势是它拥有既已建成的良好的工具生态，如PyPI与pip，其使得其他开发者很容易下载与安装您的包，而无论是一个即兴实验还是作为一个巨型软件系统的一部分。

使用这种方式共享Python代码是既已存在的传统，如果您的代码没打包到PyPI上，则它将难以被其他开发者找到并用在它们的程序中。事实上他们会怀疑这种项目是否管理不善或者已被放弃了。

像这样分发代码的消极一面是它依赖使用者理解如何安装所要求的Python版本以及会并且希望使用pip这样的工具安装您的代码及其他依赖项。虽然对于其它的开发者来说这是没问题的，但对于最终用户这并不友好。

[Python打包指南](#)提供了更多关于如何创建以及维护Python包的知识。

打包的多种方式

为了分发应用给最终用户，您应该[冻结您的应用](#)。

在Linux，您可能想会考虑[创建一个Linux分发包](#)（例 对于Debian或Ubuntu是一个 `.deb`文件）

对于Python开发者

如果您编写了一个开源的Python模块， [PyPI](#)， 更多属性参见 *The Cheeseshop*， 这是一个放它的地方。

Pip vs. easy_install

Use [pip](#). More details[here](#)

使用 [pip](#). 更多细节参见[here](#)

私人的PyPI

如果您想要从有别于PyPI的其他源安装包（也就是说，如果您的包是 **专门**（proprietary）的），您可以通过为自己开启一个服务器来建立一个这样的源，这个服务器应该开在您想共享的包所在位置的文件夹下。

例子总是有益的

作为例子，如果您想要共享一个叫做 [MyPackage.tar.gz](#) 的包，并且假设您的文件结构是这样的：

- archive
 - MyPackage
 - [MyPackage.tar.gz](#)

打开您的命令行并且输入：

```
1. $ cd archive  
2. $ python -m SimpleHTTPServer 9000
```

这运行了一个简单的http服务器，其监听端口9000并且将列出所有包（比如 **MyPackage**）。现在您可以使用任何Python包安装器从您的服务器中安装 **MyPackage**。若使用Pip，您可以这样做：

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

您的文件夹名字与您的包名相同是 **必须** 的。我曾经被这个坑过一次。但是如果您觉得创建一个叫做 [MyPackage](#) 的文件夹然后里面又有一个 [MyPackage.tar.gz](#) 文件是 **多余** 的，您可以这样共享 **MyPackage**：

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

pypiserver

[Pypiserver](#) 是一个精简的PyPI兼容服务器。它可以被用来让一系列包通过easy_install与pip进行共享。它包含一些有益的命令，诸如管理命令（[-U](#)），其可以自动更新所有它的包到PyPI上的最新版。

S3-Hosted PyPi

一个简单的个人PyPI服务器实现选项是使用Amazon S3。使用它的一个前置要求是您有一个Amazon AWS账号并且有S3 bucket。

- 安装所有您需要的东西从[PyPI](#)或者其他源
- 安装 [pip2pi](#)

- pip install git+<https://github.com/wolever/pip2pi.git>
- 跟着 **pip2pi** 的**README**文件使用**pip2tgz** 与 **dir2pi**命令
- pip2tgz packages/ YourPackage (or pip2tgz packages/ -r requirements.txt)
- dir2pi packages/

1. 上传新文件* 使用像Cyberduck这些的客户端同步整个

[packages`文件夹到您的s3 bucket](#)



- 保证您像（注意文件和路径）这样 [`packages/simple/index.html`](#) 上传了新的文件。

- **Fix**新文件许可
- 默认情况下，当您上传新文件到S3 bucket, 它们将有一个不合适的许可设置。
- 使用Amazon web console设置文件的对所有人的READ许可。
- 如果当您尝试安装一个包的时候遇上 HTTP 403 , 确保您正确设置了许可。
- 搞定
- 您可以安装您的包通过使用代码 pip install --index-url=<http://your-s3-bucket/packages/simple/> YourPackage

在Linux上分发

创建一个Linux分发包对于Linux来说是个正确的决定。

因为分发包可以不包含Python解释器，它使得下载与安装这些包可以减小2MB, [freezing your application](#).

并且，如果Python有了更新的版本，则您的应用可以自动使用新版本的Python。

`bdist_rpm`命令使得 [producing an RPM file](#)使得像Red Hat以及SuSE使用分发包变得极其简单，

无论如何，创建和维持不同配置要求给不同的发布格式（如 对于Debian/Ubuntu是.deb，而对于RedHat/Fedora是.rpm等）无疑需要大量的工作。如果您的代码是一个应用，而您计划分发到其他平台上，则您需要创建并维护各个配置要求来冻结您的应用为Windows与OSX。它比创建和维护一个单独的配置给每个平台要简单的多 [freezing tools](#)其将产生独立可执行的文件给所有Linux发布版，就像Windows与OSX上一样，

创建一个对Python版本敏感的分发包也会造成问题。可能需要告诉Ubuntu的一些版本的用户他们需要增加 [the 'dead-snakes' PPA](#)通过使用 `sudo apt-repository` 命令在他们安装您的 .deb文件，这将使用户极其厌烦。不仅如此，您会要维持每个发布版的使用指导，也许更糟的是，您的用户要去读，理解，并按它上面说的做。

下面是指导如何做上面所说事情的链接：

- [Fedora](#)
- [Debian and Ubuntu](#)
- [Arch](#)

有用的工具

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

原文: <http://pythonguidecn.readthedocs.io/zh/latest/shipping/packaging.html>

冻结 (freezing) 您的代码



“冻结”您的代码是指创建单个可执行文件，以分发给包含所有程序代码以及Python解释器的终端用户。

像“Dropbox”、“星战前夜”、“文明4”和“BitTorrent 客户端”都是如此。

进行这种分发的好处是您的用户不需要安装好所要求版本的Python（或其他）就可以运行您的应用程序。在Windows上，甚至许多Linux发行版和OS X，特定的Python版本并不总是安装好的。

此外，终端用户软件应始终是可执行的格式。以 `.py` 结尾的文件适用于软件工程师和系统管理员。

冻结的一个缺点是它会增加大约2-12MB的发行大小。另外，如果修补了Python的安全漏洞，您将负责分发更新版本的应用程序。

冻结的多种方式

[打包您的代码](#) 是指把您的库或工具分发给其他开发者。

在Linux 一个冻结的待选物是 [创建一个Linux分发包](#)（比如，对于 Debian 或 Ubuntu 是 `.deb` 文件，而对于 Red Hat 与 SuSE 是 `.rpm` 文件）

待处理

完善 “冻结您的代码” 部分 (stub)。

比较冻结工具

各解决方案的平台/特性支持性

Solution	Windows	Linux	OS X	Python 3	License	One-file mode	Zipfile import	Eggs	platform
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

从Linux到Windows的冻结只有PyInstaller支持，[其余的](#)。

注解

所有解决方案需要目前机器上安装了MS Visual C++ dll。除了py2app以外。只有Pyinstaller创建了可以自足运行的exe文件，其绑定了dll，可以传递 `-onefile` 到 `Configure.py`。

Windows

bbFreeze

前置要求是安装 [Python](#), [Setuptools](#) 以及 [pywin32](#) 的依赖项。

待处理

补充更多简单的生成 .exe的步骤。

- 安装 bbfreeze:

```
1. $ pip install bbfreeze
```

- 编写最基本的 bb_setup.py

```
1. from bbfreeze import Freezer
2.
3. freezer = Freezer(distdir='dist')
4. freezer.addScript('foobar.py', gui_only=True)
5. freezer()
```

注解

这将适用于最基本的文件脚本。 要进行更高级的冻结，您必须提供包含和排除类似路径

冻结 (freezing) 您的代码

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

- (可选) 包含图标

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

1. 为冻结器 (freezer) 提供 Microsoft Visual C 运行时 DLL。将 Microsoft Visual Studio 路径附加到您的 `sys.path` 中是可以的，但我发现在脚本所在同一文件夹中放 `msvcp90.dll` 则更加容易。

- 冻结！

```
1. $ python bb_setup.py
```

py2exe

前置要求是安装了 [Python on Windows](#)。

- 下载并且安装 <http://sourceforge.net/projects/py2exe/files/py2exe/>
- 编写 `setup.py` (配置选项清单):

```
1. from distutils.core import setup
2. import py2exe
3.
4. setup(
5.     windows=[{'script': 'foobar.py'}],
6. )
```

- (可选) 包含图标
- (可选) 单文件模式
- 生成  .exe 到 dist 目录:

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

6. 提供 Microsoft Visual C 运行时 DLL。两个选项: 在目标机器全局安装 dll 或者 与 .exe 一起分发 dll。

PyInstaller

前置是安装 [Python](#), [Setuptools](#) 以及 [pywin32](#) 依赖项。

- [更多的简单教程](#)
- [手册](#)

OS X

py2app

PyInstaller

PyInstaller可用于在Mac OS X 10.6 (Snow Leopard) 或更新版本上构建Unix可执行文件和窗口应用程序。

要安装PyInstaller，使用pip：

```
1. $ pip install pyinstaller
```

要创建标准的Unix可执行文件，使用 `script.py`：

```
1. $ pyinstaller script.py
```

这会创建，

- `script.spec` 文件，类似于 `make` 文件
- `build` 文件夹，存放日志文件
- `dist` 文件夹，存放主要的可执行文件 `script`，和一些依赖的Python库
`script.py` 会把全部内容放在同一个文件夹中。PyInstaller将所有 `script.py` 用到的Python库放到 `dist` 文件夹中。所以在分发可执行文件时，会分发整个 `dist` 文件夹。

`script.spec` 文件可以编辑成 [自定义构建](#)，比如可以：

- 将数据文件与可执行文件绑定在一起
- 包含PyInstaller无法自动推断的运行时库（`.dll` 或 `.so` 文件）
- 将Python运行时选项添加到可执行文件中

现在：代码 `script.spec` 可以用 `pyinstaller`（而不是再次使用 `script.py`）运行。

```
1. $ pyinstaller script.spec
```

要创建独立的OS X窗口应用程序，请使用 `--windowed` 选项：

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

这将在 `dist` 文件夹中创建一个代码 `script.app`。请确保在Python代码中使用GUI软件包，例如 [PyQt](#) 或[PySide](#)，来控制应用程序的图形部分。

`script.spec` 有几个与Mac OS X应用程序捆绑有关的 [选项](#)。例如，要指定应用程序的图标，请使用 `icon=path\to\icon.icns` 选项。

Linux

bbFreeze

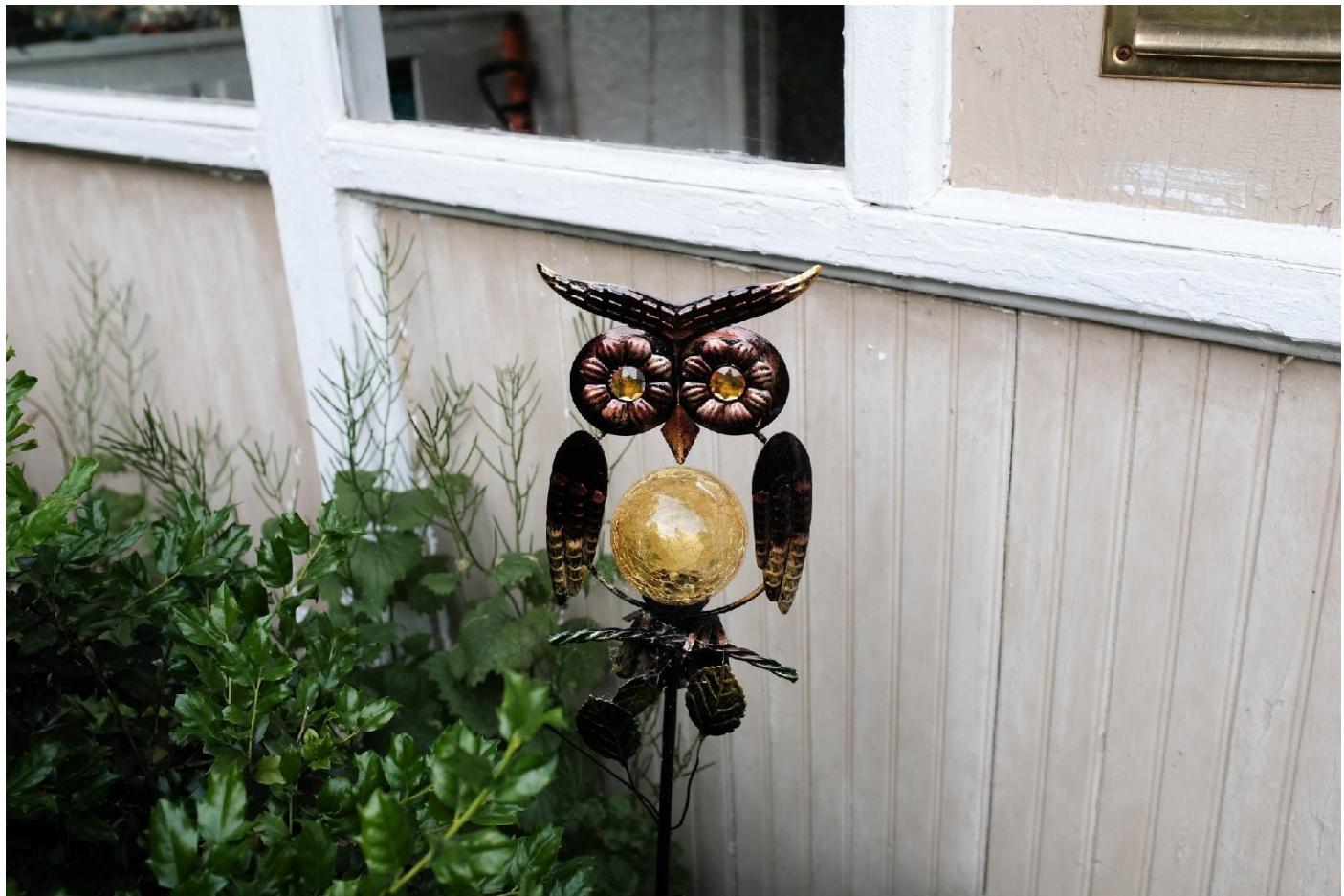
PyInstaller

原文: <http://pythonguidecn.readthedocs.io/zh/latest/shipping/freezing.html>



- [介绍](#)
- [社区](#)
- [学习Python](#)
- [文档](#)
- [新闻](#)

介绍



来自 [Python官方网站](#) 的介绍：

Python是一门通用的高级编程语言，类似于Tcl、Perl、Ruby、Scheme或者Java。其一些主要的关键特性包括：

- 非常清晰、可读的语法

Python的哲学注重可读性，从显著空格划定的代码块到使用直观关键字替代难懂符号。

- 大量的标准库和第三方模块可用于几乎任何任务

Python有时被描述为“自带电池（batteries included）”（引申为“功能完备”），就是因为它拥有大量的 [标准库](#)，这包括正则表达式、文件IO、分数处理、对象序列化等。

此外，[Python Package Index](#) 允许用户提交他们的包以得到更广泛的使用，类似于Perl的 [CPAN](#)。Python拥有生机蓬勃的社区，他们开发出非常强大的Python框架和工具，比如 [Django](#) 网络框架和 [NumPy](#) 数学库。

- 和其他系统集成

Python能够和 [Java 库](#) 集成，从而能够使用合作编程者所用的丰富Java环境。当对速度的要求变得重要时，它也可以用 [C or C++ 模块扩展](#)。

- 广泛适用于各平台

Python能够在Windows、*nix、Mac上获取到，它能运行在Java虚拟机能运行的地方，而且其参考实现CPython能

够使Python运行于C编译器能够工作的地方。

- 友好的社区

Python拥有充满生机的、庞大的 [社区](#)，他们维护着wiki、会议、无数的库、邮件列表、IRC频道（Internet Relay Chat，因特网中继聊天）等。见鬼了，他们甚至还帮助编写本指南！

关于这份指南

目的

Hitchhiker的Python指南旨在为Python初学者和专家提供一个关于Python安装、配置、和日常使用的最佳实践手册。

经由社区

本指南由 [Kenneth Reitz](#) 以开放的形式进行架构和维护。这是由社区驱动的成果，服务目标就是服务社区。

面向社区

所有水平的Python编程者对本指南所有的贡献都是欢迎的。如果您觉得本指南没有覆盖到某些内容，请在GitHub上 [fork](#) 指南，并提交一个pull请求。

欢迎每个人的贡献，不管是高手还是新手。如果您对提交的合适性、完成度或者准确度方面有任何疑问，本指南的作者会非常愿意帮助您。

想要开始为Hitchhiker的指南做贡献，请见 [贡献](#) 页面。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/introduction.html>

社区



BDFL

Python的创始人Guido van Rossum 通常被称为BDFL — 生活的仁慈独裁者。

Python软件基金会

Python软件基金会的任务是促进、保护和发展Python编程语言，同时支持和帮助Python编程者的多样、国际化的社区的成长。

[了解更多关于PSF的内容 .](#)

PEPs

PEPs是 Python 增强提议。它们描述了Python自身或者相关库的变化。

这里有三种不同的PEPs类型（由 [PEP 1 定义](#)）：

Standards

描述一项新特性或者实现。

Informational

描述一个设计issue、大体方针、或者社区信息。

Process

描述和Python相关的过程。

著名的 PEPS

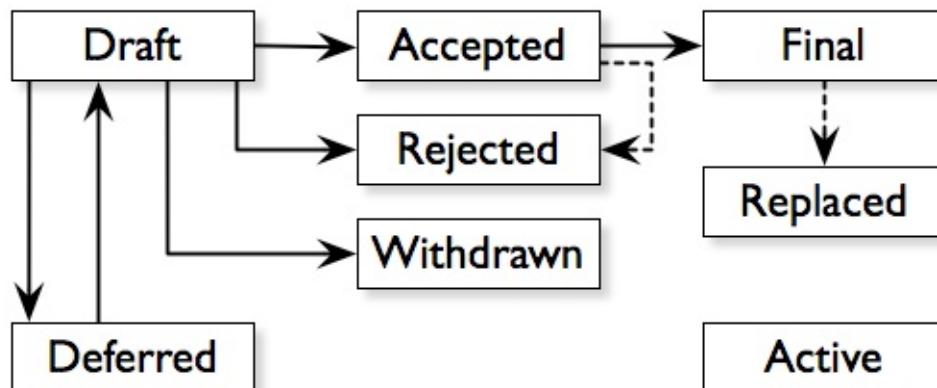
这里有少许PEPs可以认为是要求阅读的：

- [PEP 8](#): Python风格指南。完整阅读它，并照着做。
 - [PEP 20](#): Python之禅。一个19行的小诗，简述了Python背后的哲学。
 - [PEP 257](#): 文档字符串约定。给出了与Python文档字符串相关的语义和约定的指导方针。
- 您可以在 [PEP 索引](#) 上了解更多。

提交一个 PEP

PEPs会被同行审阅，并在大量的讨论后决定接受或拒绝。任何人都能编写和提交一个PEP供以审阅。

这里有一个接受PEP的工作流：



Python会议

Python社区主要的事件就是开发者会议。两个最著名的会议就是在美国举办的PyCon，和在欧洲举办的姊妹版会议EuroPython。

会议的综合列表维护在 pycon.org 上。

Python用户组

用户组就是一群Python开发者见面现身或讨论感兴趣的Python话题的地方。一系列本地用户组维护在 [Python 软件基金会 Wiki](#) 中。

原文: <http://pythonguidecn.readthedocs.io/zh/latest/intro/community.html>

学习Python



初学者

The Python Tutorial

这是官方教程。它涵盖了所有的基础，并提供了Python语言和标准库的浏览。推荐给需要快速开始的人。

[The Python Tutorial](#)

Real Python

Real Python 是由各种专业的Python开发团队创建的免费且深入的Python教程库。在Real Python中，您可以从头开始学习Python的所有内容。从Python的最基础的知识到Web开发和Web爬取，再到数据可视化等等。

[Real Python](#)

Python Basics

pythonbasics.org 是初学者的入门教程，包括练习、涵盖了基础知识，还有深入的课程（如面向对象编程和正则

表达式)。

Python basics

Python for beginners

thePYTHONGURU.COM是一个专注于初学程序员的教程。它深入涵盖了许多python概念。它还教您一些Python的高级概念，如lambda表达式、正则表达式等。最后，它以教程“如何使用python访问MySQL数据库”结束。

Python for beginners

Learn Python Interactive Tutorial

Learnpython.org是一个简单易懂的了解Python的途径。网站采用了和流行的Try Ruby相同的方式，有一个交互式的Python解释器内建于网站中，允许您在学习Python课程时不需要在本地安装Python。

Learn Python

如果您想要更加传统的书籍，*Python For You and Me* 是一个极佳的学习Python所有方面的资源。

Python for You and Me <<https://pymbook.readthedocs.io/>>;

Learn Python Step by Step

Techbeamers.com 提供了手把手的教程讲解Python。每个教程都补充了相关代码段，并配备了关于所学主题的后续测验。里面的 Python 面试题部分可以帮助求职者。您还可以阅读必备的 Python 技巧，并学习编写高质量代码的最佳编程实践。在这里，你将获得快速学习Python的正确平台。

学习 Python 从基础到高级

Online Python Tutor

Online Python Tutor在视觉上向您一步步展示程序是如何运行的。Python Tutor帮助人们克服学习编程的基本障碍，让您明白程序源码中的每一行的执行会有怎样的结果。

Online Python Tutor

Invent Your Own Computer Games with Python

这个新手书籍是面向没有任何编程经验的对象。每个章节都有一个小游戏的源码，这些程序例子说明了编程概念，让读者了解到程序是什么样的。

Invent Your Own Computer Games with Python

Hacking Secret Ciphers with Python

本书向完全的新手讲解了Python编程和基本的密码学知识。所有章节提供了多种加密源码，也提供了破解程序。

[Hacking Secret Ciphers with Python](#)

Learn Python the Hard Way

这是一部极佳的新手编程者的Python指南。它涵盖了从控制台到web的“hello world”。

[Learn Python the Hard Way](#)

Crash into Python

这个指南也叫 *Python for Programmers with 3 Hours*，它给有其他语言开发经验的开发人员一个关于Python的快速教程。

[Crash into Python](#)

Dive Into Python 3

Dive Into Python 3对准备使用Python 3的人来说是一本不错的书。如果您是从Python 2迁移到Python 3或者您已经有其他语言的编程经验，那么将会很好阅读。

[Dive Into Python 3](#)

Think Python: How to Think Like a Computer Scientist

Think Python 通过使用Python语言来介绍计算机科学中的基本概念。它着重提供大量的联系、最少的术语以及每章一个小节的调试部分。

在探索Python语言不通的特性时，作者编织了不同的设计模式和最佳实践。

本书中还包括一些案例研究，将书中的主题应用到实际例子中，供读者对主题进行更深入的讨论。案例研究包括GUI和Markov分析的任务。

[Think Python](#)

Python Koans

Python Koans是Edgecase's Ruby Koans的迁移版。它使用测试驱动的方法（参阅TEST DRIVEN DESIGN SECTION）提供一个交互式的教程，来讲解基本的Python概念。通过修复在测试脚本中失败的断言，从而提供连续的步骤来学习Python。

对于那些使用语言并找出自己的困惑的人来说，这会是个有趣并有吸引力的选择。对于新手来说，拥有一个额外的资源或者参考会是很有用的。

[Python Koans](#)

要了解更多关于测试驱动开发的内容，可以查看以下资源：

测试驱动开发

A Byte of Python

一本为新手讲解Python的免费入门书籍，它假定读者没有编程经验。

[A Byte of Python for Python 2.x](#) [A Byte of Python for Python 3.x](#)

Learn to Program in Python with Codecademy

此Codecademy课程面向绝对的Python初学者。这门免费、互动的课程提供和教授Python编程的基础（和以后）的内容，同时测试用户之间知识的进展。本课程还内置了一个解释器，用于获取学习过程中的即时反馈。

[Learn to Program in Python with Codecademy](#)

Code the blocks

Code the blocks 为初学者提供免费的交互式Python教程。它将Python编程与3D环境相结合，您可以在其中“放置方块”并构建结构。本教程将教您如何使用Python创建逐渐精细的3D结构，从而使得学习Python的过程变得有趣和吸引人。

[Code the blocks](#)

中级

Python Tricks: The Book

通过简单的示例发现Python的最佳实践，并开始编写更美观的且Pythonic的代码。“Python Tricks: The Book”向您展示了具体方法。

您将通过实际示例和清晰的叙述掌握Python的中级和高级功能：

[Python Tricks: The Book](#)

Effective Python

本书包含59种具体方法来改进编写Pythonic代码。在这227页中，这是一个关于程序员需要做的一些最常见的程序以成为高效的中级Python程序员的非常简要的概述。

[Effective Python](#)

进阶者

Pro Python

本书是面向从中级到高级，想明白Python是如何以及为何这样工作，如何将代码水平提高一级的Python程序员。

Pro Python

Expert Python Programming

Expert Python Programming讲解编写Python的最佳实践，并专注更高级的人员。

它以诸如装饰器（伴随缓存、代理、上下文管理器、案例研究）、方法解析顺序、使用super()、元编程和一般 [PEP 8](#) 上的最好实践开始。

它有一个详细的、多章的关于编写的案例研究，发行了一个包并最终成为一个应用，包含使用zc.buildout的一个章节。后面的章节详细讲述了最佳实践，比如编写文档、测试驱动开发、版本控制、优化和分析。

Expert Python Programming

A Guide to Python's Magic Methods

这是Rafe Kettler发表博文的集合，解释了Python中的“魔法方法”。魔法方法由双下划线包围（比如 `__init__`），能够使类和对象表现出不同的、魔法的行为。

A Guide to Python's Magic Methods

注解

Rafekettler.com目前已关闭，您可以直接访问他们的Github版本。 在这里您可以找到一个PDF版本：[A Guide to Python's Magic Methods \(repo on GitHub\)](#)

工程师和科学家

A Primer on Scientific Programming with Python

A Primer on Scientific Programming with Python 由 Hans Petter Langtangen 编写，主要涵盖了 Python 在科学领域的使用。在这本书中，例子是从数学和自然科学中选出的。

A Primer on Scientific Programming with Python

Numerical Methods in Engineering with Python

Numerical Methods in Engineering with Python 由 Jaan Kiusalaas 编写，其重点是数值方法以及如何用Python来实现。

Numerical Methods in Engineering with Python

各种各样的话题

Problem Solving with Algorithms and Data Structures

Problem Solving with Algorithms and Data Structures涵盖了一系列数据结构和算法。所有概念都用Python代码说明，提供了可在浏览器中直接运行的交互式样例。

[Problem Solving with Algorithms and Data Structures](#)

Programming Collective Intelligence

Programming Collective Intelligence介绍了大量基础的机器学习和数据挖掘方法。其展示在数学上并不是很正式，而是更侧重于解释潜在的直觉，以及展示如何使用Python来实现算法。

[Programming Collective Intelligence](#)

Transforming Code into Beautiful, Idiomatic Python

Transforming Code into Beautiful, Idiomatic Python 是由 Raymond Hettinger制作的视频。通过它可以学习到更好地使用Python最佳特性，通过一系列的代码转换来改进现有代码，“当您看见这个，就那样去做”。

[Transforming Code into Beautiful, Idiomatic Python](#)

Fullstack Python

Fullstack Python为使用Python进行Web开发提供了完整的自顶向下的资源。

范围涵盖从设置Web服务器到设计前端、选择数据库、优化/缩放等。

顾名思义，它涵盖了从头开始构建和运行完整的Web应用程序所需的所有内容。

[Fullstack Python](#)

参考

Python in a Nutshell

Python in a Nutshell 由 Alex Martelli编写，涵盖了Python跨平台的多数用法，从它的语法到内建库，再到比如说编写C扩展的高级主题。

[Python in a Nutshell](#)

The Python Language Reference

这是Python的参考手册，它涵盖了这门语言的语法和核心语义。

The Python Language Reference

Python Essential Reference

Python Essential Reference，由David Beazley撰写，是Python的最终参考指南。它简明扼要地解释了标准库的核心语言和最重要的部分。它涵盖了Python 3和2.6版本。

Python Essential Reference

Python Pocket Reference

Python Pocket Reference 由 Mark Lutz 编写，是一个了解核心语言的易于使用的参考，介绍了常用的模块和工具集。它涵盖了Python 3 和 Python 2。

Python Pocket Reference

Python Cookbook

Python Cookbook 由 David Beazley 和 Brian K. Jones 编写，打包了许多具有实践意义的“食谱”。这本书涵盖了核心Python语言，也涵盖了诸多不同应用的常见任务。

Python Cookbook

Writing Idiomatic Python

Writing Idiomatic Python 由 Jeff Knupp 编写，包含了最常见和最重要的Python习语，其形式尽可能地有辨识度和易于理解。每个习语都是编写一些常用代码片段的推荐方式，其后会解释为什么这个习语是重要的。每个习语均有两个代码样例：“有害的”方式和“理想的”方式。

For Python 2.7.3+For Python 3.3+

原文：<http://pythonguidecn.readthedocs.io/zh/latest/intro/learning.html>

文档



官方文档

Python语言和库的官方文档能够在这里找到：

- [Python 2.x](#) - [Python 3.x](#)

Read the Docs

Read the Docs是一个流行的社区项目，存放着开源软件的文档。它拥有很多Python模块，优秀且流行。

[Read the Docs](#)

pydoc

pydoc 是一个在您安装Python时跟着安装的工具。它允许您在shell中快速检索和查找文档。比如，如果您需要对 `time` 模块的进行快速回顾，查看文档就是像下面这么简单：

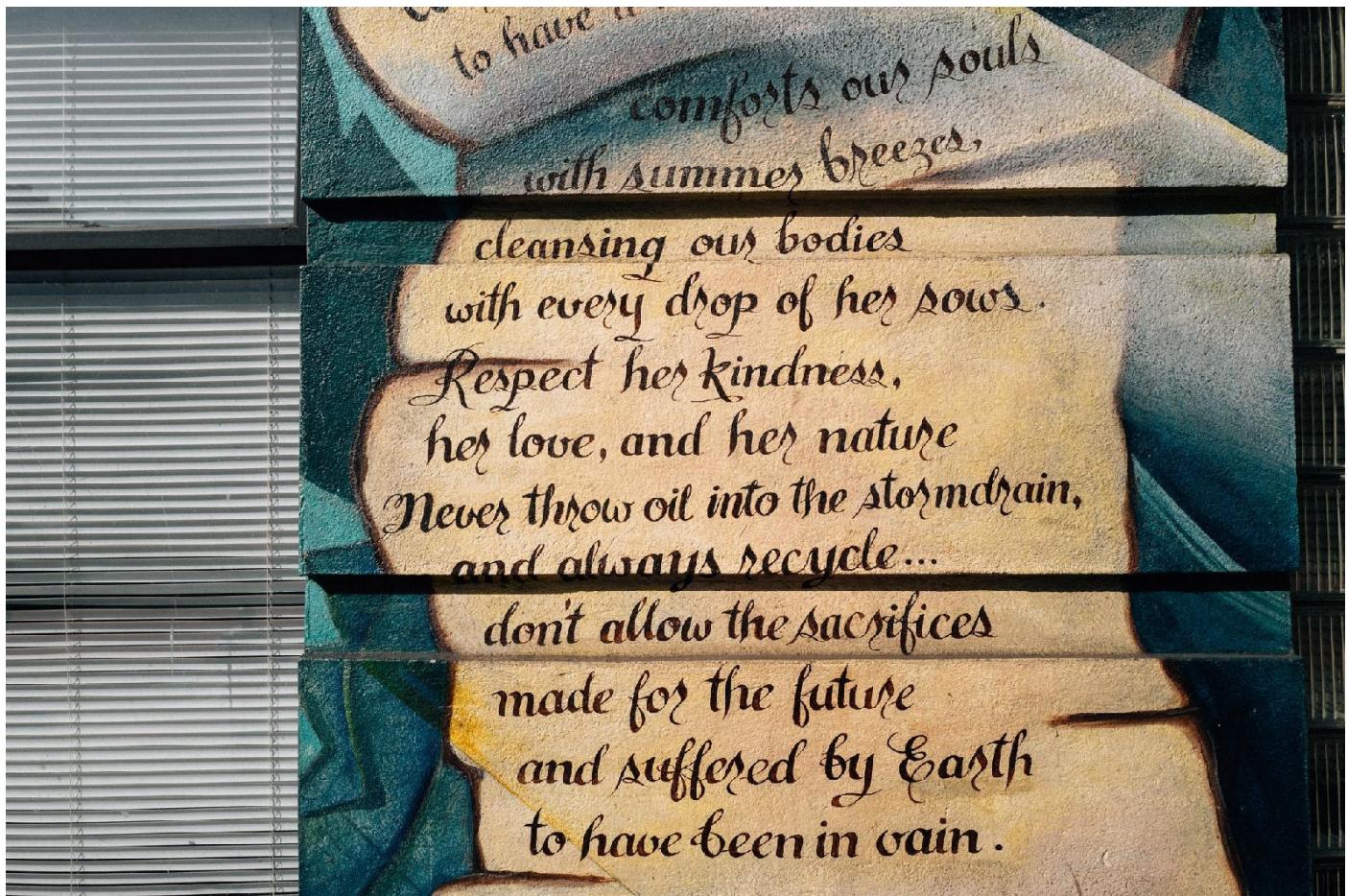
```
1. $ pydoc time
```

上面的命令和打开Python REPL然后运行下面指令是基本等价的：

```
1. >>> help(time)
```

原文：<http://pythonguidecn.readthedocs.io/zh/latest/intro/documentation.html>

新闻



Planet Python

这是一个聚合的，展示不断增长的开发者内容的Python新闻。

[Planet Python](#)

/r/python

/r/python是Reddit的Python社区，在这里用户会贡献和投票出和Python相关的新闻。

[/r/python](#)

Talk Python Podcast

第一个关注 Python 的播客，涵盖了 Python 中的人和想法。

[Talk Python To Me](#)

Python Bytes Podcast

一个简短的 Python 播客，涵盖了最近的开发人员头条。

Python Bytes

Pycoder's Weekly

Pycoder's Weekly是一个面向Python开发者的免费的每周简讯，包括项目、文章、新闻和工作。

Pycoder's Weekly

Python Weekly

Python Weekly是一个免费的每周简讯，其特色包括与Python相关的新闻、文章、新的发行版本、工作等。

Python Weekly

Python News

Python News是Python官方网站的新闻章节。它简短突出来自Python社区的新闻。

Python News

Import Python Weekly

Weekly Python Newsletter包含有Python文章、项目、视频、Tweets，并发送到您的收件箱。能够保持您的Python编程技能持续更新。

Import Python Weekly Newsletter

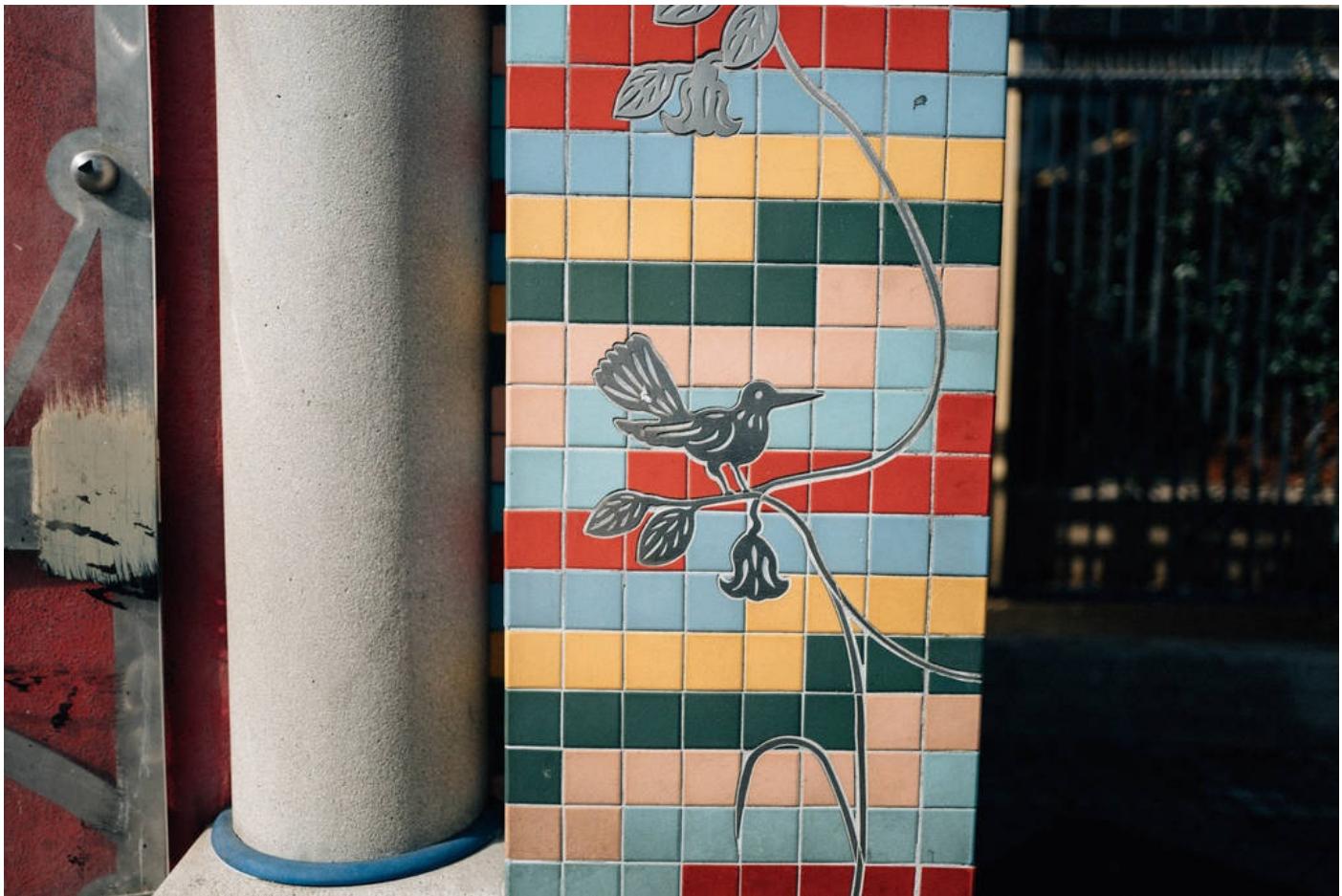
Awesome Python Newsletter

每周最流行的Python新闻、文章和包的概述。

Awesome Python Newsletter

原文：<http://pythonguidecn.readthedocs.io/zh/latest/intro/news.html>

贡献



Python-guide目前还在开发中，欢迎大家贡献代码。

如果您有feature request, 修改建议, 或者bug报告, 请在 [GitHub](#) 上新建issue。如果想要提交补丁, 请直接 pull request到 [GitHub](#)。一旦您的修改被merge, 您会被自动添加到[贡献者一览](#) 中。

风格指南

请戳这里 [风格指南指引](#) .

要做的事

如果您想为我们做些贡献, 这里有我们的计划。一个简短的 [todo](#) 列表。

```
- Establish "use this" vs "alternatives are...." recommendations待处理Write about Blueprint(原始记录 见 /home/docs/checkouts/readthedocs.org/userbuilds/pythonguidecn/checkouts/latest/docs/scenarios/admin.rst, 第 328 行。)待处理完善 "冻结您的代码" 部分 (stub)。([原始记录] (http://pythonguidecn.readthedocs.io/zh/latest/shipping/freezing.html#index-0) 见 /home/docs/checkouts/readthedocs.org/user_builds/pythonguidecn/checkouts/latest/docs/shipping/freezing.rst, 第 29 行。)待处理补充更多简单的生成 .exe的步骤。([原始记录] (http://pythonguidecn.readthedocs.io/zh/latest/shipping/freezing.html#index-1) 见 /home/docs/checkouts/readthedocs.org/user_builds/pythonguidecn/checkouts/latest/docs/shipping/freezing.rst, 第 65 行。)待处理包括每个列出项目中典型代码的例子。解释为什么它是非常优秀的代码, 举出较复杂的例子。([原始记录]
```

(<http://pythonguidecn.readthedocs.io/zh/latest/writing/reading.html#index-0>) 见
/home/docs/checkouts/readthedocs.org/user_builds/pythonguidecn/checkouts/latest/docs/writing/reading.rst, 第 39
行。)待处理解释快速识别数据结构, 算法, 并确定代码内容的技术。([原始记录_]
(<http://pythonguidecn.readthedocs.io/zh/latest/writing/reading.html#index-1>) 见
/home/docs/checkouts/readthedocs.org/user_builds/pythonguidecn/checkouts/latest/docs/writing/reading.rst, 第 41
行。)

原文: <http://pythonguidecn.readthedocs.io/zh/latest/notes/contribute.html>

许可证



本指南基于 Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license <<https://creativecommons.org/licenses/by-nc-sa/3.0/>>; 的许可。

原文：<http://pythonguidecn.readthedocs.io/zh/latest/notes/license.html>

风格指南指引



所有文档都有一致的格式，以帮助更好地理解文档。为了使指导更容易消化，所有贡献都应适应风格指南的规则。

本指南以 [reStructuredText](#) 形式编写。

注解

本指南部分内容可能尚未符合本指南指引。欢迎更新这些部分以保持同步。

注解

在任何一个渲染后的HTML页面，您可以点击“[查看源码（Show Source）](#)”来看作者是如何排版的。

关联

尽量保持任何贡献与 [本指南目的](#) 相关。

- 避免在主题中包含太多与Python开发并不直接相关的信息。
- 如果其他资源已存在，最好以链接的形式来展示。确保描述出您所链接的内容和原因。
- [Cite](#)引用在需要的地方。
- 如果某主题并不与Python直接相关，但是和Python之间的关联又很有用（比如Git、GitHub、数据库等），以链接的形式引用这些资源，并描述为什么它对Python有用。
- 如果疑问，就去询问。

标题

使用下列风格作为标题。

章节标题：

- ```
1. #####
2. 章节 1
3. #####
```

页面标题：

- ```
1. ======  
2. 时间是种幻觉  
3. ======
```

小节标题：

1. 午餐时间加倍

次小节标题：

- ```
1. 非常深
2. ~~~~~~
```

# 换行

每78个字符进行文字换行。必要时可以超过78个字符，尤其是那种换行使得源内容更难阅读的情况。

序列逗号 ([serial comma](#)) (也称为Oxford comma, 牛津逗号) 的使用是100%没有选择的。任何尝试以缺少的连续逗号提交内容将导致该项目的永久性移除，因为完全缺乏品味。

流放？您在开玩笑吗？希望我们永远不必找出来。

# 代码例子

所有代码示例要在70个字符进行换行，以避免出现水平滚动条。

命令行例子：

- ```
1. ... code-block:: console  
2.  
3.     $ run command --help  
4.     $ ls ..
```

确保每行前面包含了  前缀。

Python解释器例子：

```
1. Label the example::
2.
3. ... code-block:: python
4.
5.     >>> import this
```

Python 例子：

```
1. Descriptive title::
2.
3. ... code-block:: python
4.
5.     def get_answer():
6.         return 42
```

外部链接

- 链接时最好使用众所周知的主题（比如一些合适的名词）：

Sphinx_ 通常用来文档化Python。

[.. _Sphinx: http://sphinx.pocoo.org](#)

- 最好使用带有内联链接的描述性标签，而不是单纯的链接：

阅读 [Sphinx 教程 <http: sphinx.pocoo.org="" tutorial.html="">](#) —

- 避免使用诸如“点击这里”、“这个”等标签。最好使用描述性标签（值得搜索引擎优化，SEO worthy）。

指向指南内部章节的链接

要交叉引用本文档的其他部分，使用  关键字和标签。

要使引用标签更加清晰和独特，通常加上一个  后缀：

```
1. ... _some-section-ref:
2.
3. Some Section
4. -----
```

注意和警告

使用适当的 **警告指示** 来说明注意内容。

注意：

1. ... note::
2. 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.

警告：

1. ... warning:: DON'T PANIC

要做的事

请用 **todo 指示** 来标记本指南中任何未完成的部分。避免使 **要做的事** 混乱，为未完的文档或者大量未完的小节使用单独的 **todo**。

1. ... todo::
2. Learn the Ultimate Answer to the Ultimate Question
3. of Life, The Universe, and Everything

原文：<http://pythonguidecn.readthedocs.io/zh/latest/notes/styleguide.html>