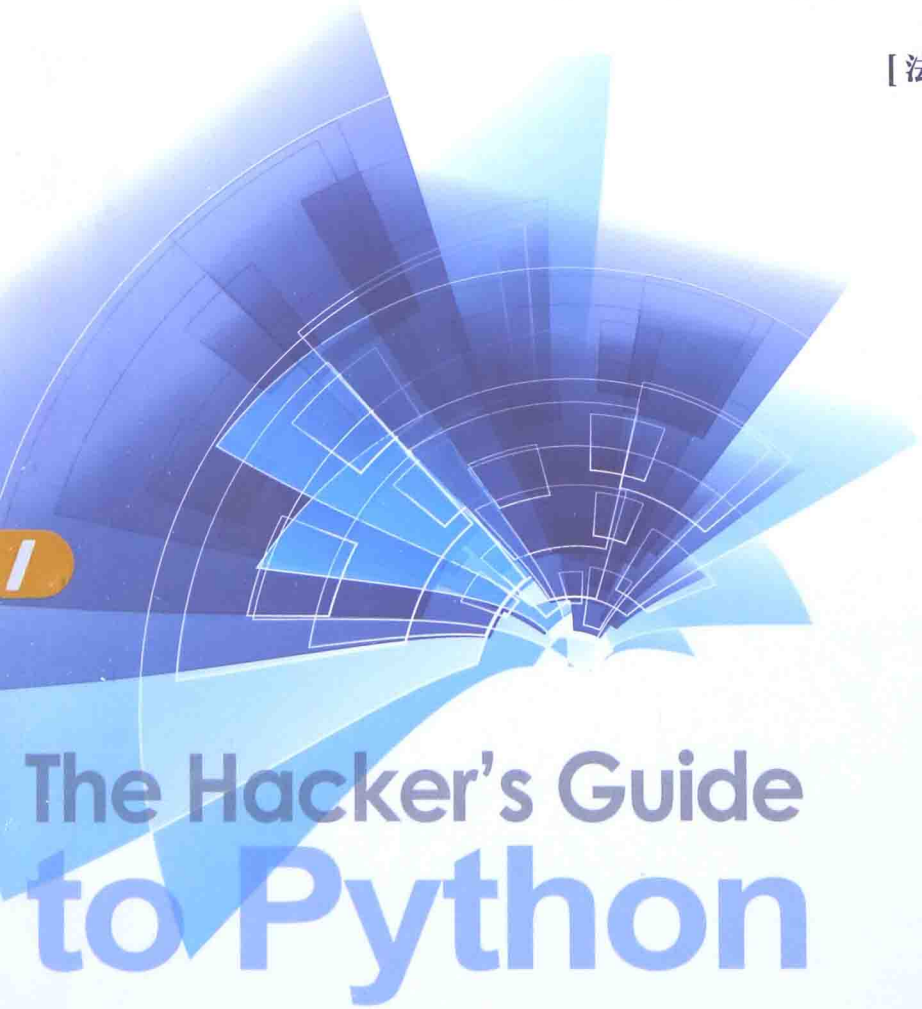
Python 高手之路

[法] Julien Danjou 著
王飞龙 译

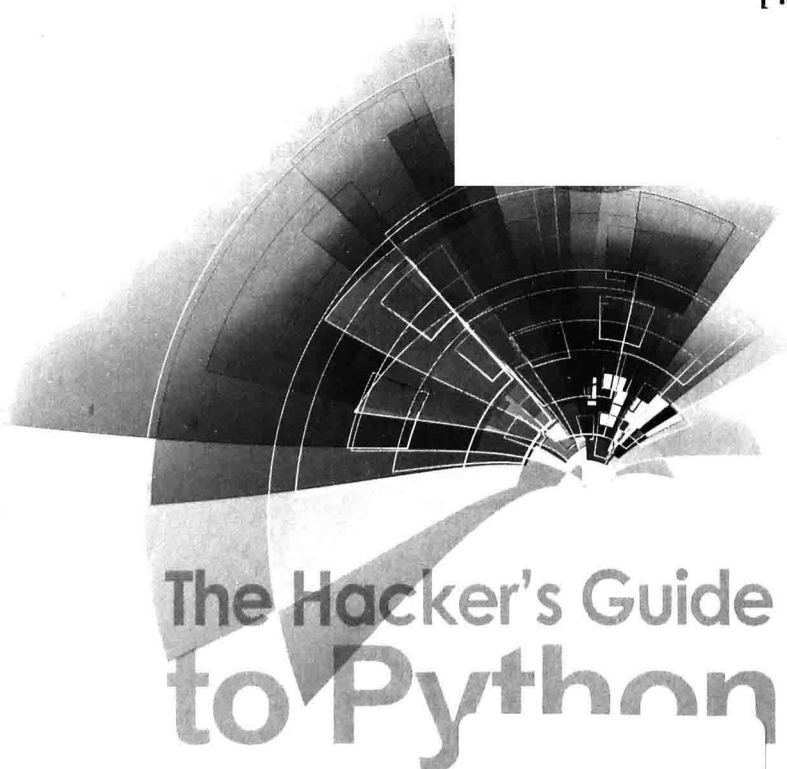


The Hacker's Guide to Python

人民邮电出版社
POSTS & TELECOM PRESS

Python 高手之路

[法] Julien Danjou 著
王飞龙 译



人民邮电出版社
北京

图书在版编目（CIP）数据

Python高手之路 /（法）丹乔（Danjou, J.）著 ；王飞龙译. -- 北京 ：人民邮电出版社，2015.5
书名原文：The hacker's guide to Python
ISBN 978-7-115-38713-4

Ⅰ. ①P… Ⅱ. ①丹… ②王… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2015)第065975号

内 容 提 要

这不是一本常规意义上 Python 的入门书。这本书中没有 Python 关键字和 `for` 循环的使用，也没有细致入微的标准库介绍，而是完全从实战的角度出发，对构建一个完整的 Python 应用所需掌握的知识进行了系统而完整的介绍。更为难得的是，本书的作者是开源项目 OpenStack 的 PTL（项目技术负责人）之一，因此本书结合了 Python 在 OpenStack 中的应用进行讲解，非常具有实战指导意义。

本书从如何开始一个新的项目讲起，首先是整个项目的结构设计，对模块和库的管理，如何编写文档，进而讲到如何分发，以及如何通过虚拟环境对项目进行测试。此外，本书还涉及了很多高级主题，如性能优化、插件化结构的设计与架构、Python 3 的支持策略等。本书适合各个层次的 Python 程序员阅读和参考。

◆ 著　[法] Julien Danjou
译　王飞龙
责任编辑　杨海玲
责任印制　张佳莹　焦志炜
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷
◆ 开本：800×1000　1/16
印张：13
字数：277 千字　2015 年 5 月第 1 版
印数：1-3 000 册　2015 年 5 月北京第 1 次印刷
著作权合同登记号　图字：01-2014-5625 号

定价：49.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

中文版序

亲爱的中国读者你们好！

祝贺你，你正在读 *The Hacker's Guide to Python* 一书的中文版。我非常高兴看到这本书最终翻译完成，这样你就可以用自己的语言去阅读。这是这本书第三种语言的版本（之前已经有了英语和韩语两个版本）。能够有更多的读者看到这本书真是太棒了！

你将阅读的这本书的大部分内容来自我在 OpenStack 这个大规模项目中开发 Python 代码时的经验。你们是非常幸运的，因为这本书是由王飞龙翻译的，他是一名软件工程师，他和我同在 OpenStack 社区做开发工作。因此，高质量的翻译和对本书内容的精确表述是可期待的，因为飞龙对本书涉及的内容有着很好的理解。

真心希望你们能喜欢这本书。祝你阅读愉快！

Julien Danjou

前言

如果你读到这里，你肯定已经使用 Python 有一阵子了。你可能是通过一些文档学习的，钻研了一些已有的项目或者从头开发，但不管是哪种情况，你都已经在以自己的方式学习它了。直到两年前我加入 OpenStack 项目组之前，这其实也正是我个人熟悉 Python 的方法。

在此之前，我只是开发过一些“车库项目[①]”级别的 Python 库或应用程序，而一旦你参与开发涉及数百名开发人员并有着上万个用户的软件或库时，情况就会有所不同。OpenStack 平台有超过 150 万行 Python 代码，所有代码都需要精确、高效，并根据用户对云计算应用程序的需求进行任意扩展。在有了这一规模的项目之后，类似测试和文档这类问题就一定需要自动化，否则根本无法完成。

我刚开始加入 OpenStack 的时候，我认为自己已经掌握了不少 Python 知识，但这两年，在我起步时无法想象其规模的这样一个项目上，我学到了更多。而且我还有幸结识了很多业界最棒的 Python 黑客，并从他们身上获益良多——大到通用架构和设计准则，小到各种有用的经验和技巧。通过本书，我想分享一些我所学到的最重要的东西，以便你能构建更好的 Python 应用，并且是更加高效地构建。

① 作者这里的意思是规模很小，比较业余的项目。——译者注

目录

第1章 项目开始

1.1 Python 版本

你很可能会问的第一个问题就是：“我的软件应该支持 Python 的哪些版本？”这是一个好问题，因为每个 Python 新版本都会在引入新功能的同时弃用一些老的功能。而且，Python 2.x 和 Python 3.x 之间有着**巨大**的不同，这两个分支之间的剧烈变化导致很难使代码同时兼容它们。本书后面章节会进一步讨论，而且当刚刚开始一个新项目时很难说哪个版本更合适。

- 2.5 及更老的版本目前基本已经废弃了，所以不需要再去支持它们。如果实在想支持这些更老的版本，要知道再让程序支持 Python 3.x 会更加困难。如果你确实有可能会遇到一些安装了 Python 2.5 的老系统，那真没什么好办法。
- 2.6 版本在某些比较老的操作系统上仍然在用，如 Red Hat 企业版 Linux（Red Hat Enterprise Linux）。同时支持 Python 2.6 版本和更新的版本并不太难，但是，如果你认为自己的程序不太可能会在 2.6 版本上运行，那就没必要强迫自己支持它。
- 2.7 版本目前是也将仍然是 Python 2.x 的最后一个版本。将其作为主要版本或主要版本之一来支持是正确的选择，因为目前仍然有很多软件、库和开发人员在使用它。Python 2.7 将被继续支持到 2020 年左右，所以它很可能不会很快消失。
- 3.0、3.1 和 3.2 版本在发布后都被快速地更替，并没有被广泛采用。如果你的代码已经支持了 2.7 版本，那么再支持这几个版本的意义并不大。
- 3.3 和 3.4 版本都是 Python 3 最近发行的两个版本，也是应该重点支持的版本。Python 3.3 和 3.4 代表着这门语言的未来，所以除非正专注于兼容老的版本，否则都应该先确保代码能够运行在这两个最新的版本上。

总之，在确实有需要的情况下支持 2.6 版本（或者想自我挑战），必须支持 2.7 版本，如果需要保证软件在可预见的未来也能运行，就需要也支持 3.3 及更高的版本。忽略那些更老的 Python 版本基本没什么问题，尽管同时支持所有这些版本是有可能的：CherryPy 项目（http://cherrypy.org）支持 Python 2.3 及所有后续版本（http://docs.cherrypy.org/stable/intro/install.html）。

编写同时支持 Python 2.7 和 3.3 版本的程序的技术将在第 13 章介绍。某些技术在后续的示例代码中也会涉及，所有本书中的示例代码都同时支持这两个主要版本。

1.2　项目结构

项目结构应该保持简单，审慎地使用包和层次结构，过深的层次结构在目录导航时将如同梦魇，但过平的层次结构则会让项目变得臃肿。

一个常犯的错误是将单元测试放在包目录的外面。这些测试实际上应该被包含在软件的子一级包中，以便：

- 避免被 **setuptools**（或者其他打包的库）作为 tests 顶层模块自动安装；
- 能够被安装，且其他包能够利用它们构建自己的单元测试。

图 1-1 展示了一个项目的标准的文件层次结构。

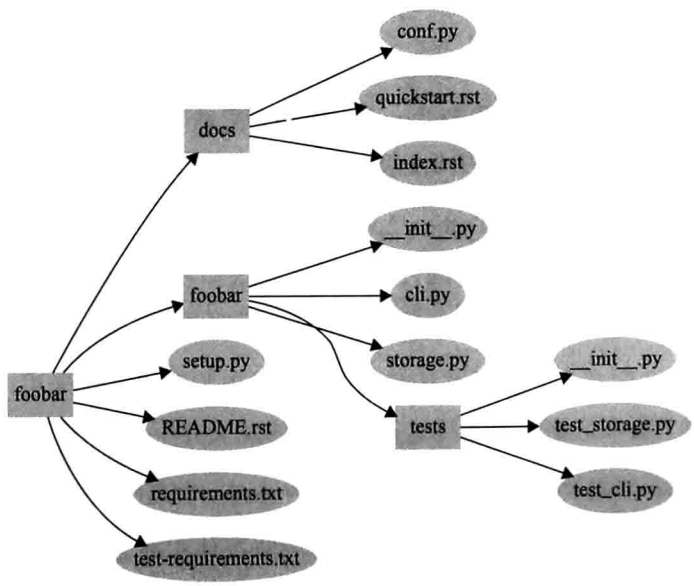


图 1-1　标准的包目录结构

setup.py 是 Python 安装脚本的标准名称。在安装时，它会通过 Python 分发工具（distuils）进行包的安装。也可以通过 README.rst（或者 README.txt，或者其他合适的名字）为用户提供重要信息。requirements.txt 应该包含 Python 包所需要的依赖包，也就是说，所有这些包都会预先通过 pip 这样的工具进行安装以保证你的包能正常工作。还可以包含 test-requirements.txt，它应该列出运行测试集所需要的依赖包。最后，docs 文件夹应该包括 reStructuredText 格式的文档，以便能够被 Sphinx 处理（参见 3.1 节）。

包中还经常需要包含一些额外的数据，如图片、shell 脚本等。不过，关于这类文件如何存放并没有一个统一的标准。因此放到任何觉得合适的地方都可以。

下面这些顶层目录也比较常见。

- etc 用来存放配置文件的样例。
- tools 用来存放与工具有关的 shell 脚本。
- bin 用来存放将被 setup.py 安装的二进制脚本。
- data 用来存放其他类型的文件，如媒体文件。

一个常见的设计问题是根据将要存储的代码的类型来创建文件或模块。使用 functions.py 或者 exceptions.py 这样的文件是很**糟糕的**方式。这种方式对代码的组织毫无帮助，只能让读代码的人在多个文件之间毫无理由地来回切换。

此外，应该避免创建那种只有一个__init__.py 文件的目录，例如，如果 hooks.py 够用的话就不要创建 hooks/__init__.py。如果创建目录，那么其中就应该包含属于这一分类/模块的多个 Python 文件。

1.3 版本编号

可能你已经有所了解，Python 生态系统中正在对包的元数据进行标准化。其中的一项元数据就是*版本号*。

PEP 440（http://www.python.org/dev/peps/pep-0440/）针对所有的 Python 包引入了一种版本格式，并且在理论上所有的应用程序都应该使用这种格式。这样，其他的应用程序或包就能简单而可靠地识别它们需要哪一个版本的包。

PEP440 中定义版本号应该遵从以下正则表达式的格式：

```
N[.N]+[{a|b|c|rc}N][.postN][.devN]
```

它允许类似 1.2 或 1.2.3 这样的格式，但需注意以下几点。

- 1.2 等于 1.2.0，1.3.4 等于 1.3.4.0，以此类推。
- 与 *N[.N]+*相匹配的版本被认为是**最终版本**。
- 基于日期的版本（如 2013.06.22）被认为是无效的。针对 PEP440 格式版本号设计的一些自动化工具，在检测到版本号大于或等于 1980 时就会抛出错误。

最终即将发布的组件也可以使用下面这种格式。

- *N[.N]+aN*（如 1.2a1）表示一个 **alpha 版本**，即此版本不稳定或缺少某些功能。
- *N[.N]+bN*（如 2.3.1b2）表示一个 **beta 版本**，即此版本功能已经完整，但可能仍有 bug。
- *N[.N]+cN* 或 *N[.N]+rcN*（如 0.4rc1）表示**候选版本**（常缩写为 RC），通常指除非有重大的 bug，否则很可能成为产品的最终发行版本。尽管 *rc* 和 *c* 两个后缀含义相同，但如果二者同时使用，*rc* 版本通常表示比 *c* 更新一点。

通常用到的还有以下这些后缀。

- *.postN*（如 1.4.post2）表示一个**后续版本**。通常用来解决发行过程中的细小问题（如发行文档有错）。如果发行的是 bug 修复版本，则不应该使用*.postN* 而应该增加小的版本号。
- *.devN*（如 2.3.4.dev3）表示一个**开发版本**。因为难以解析，所以这个后缀并不建议使用。它表示这是一个质量基本合格的发布前的版本，例如，2.3.4.dev3 表示 2.3.4 版本的第三个开发版本，它早于任何的 alpha 版本、beta 版本、候选版本和最终版本。

这一结构可以满足大部分常见的使用场景。

注意

你可能已经听说过语义版本（http://semver.org/），它对于版本号提出了自己的规则。这一规范和 PEP 440 部分重合，但二者并不完全兼容。例如，语义版本对于预发布版本使用的格式 *1.0.0.-alpha+001* 就与 PEP 440 不兼容。

如果需要处理更高级的版本号，可以考虑一下 PEP 426（http://www.python.org/dev/peps/pep-0426）中定义的**源码标签**，这一字段可以用来处理任何版本字符串，并生成同 PEP 要求一致的版本号。

许多分布式版本控制系统（Distributed Version Control System，DVCS）平台，如 Git 和 Mercurial，都可以使用唯一标识的散列字符串[①]作为版本号。但遗憾的是，它不能与 PEP 440 中定义的模式兼容：问题就在于，唯一标识的散列字符串不能排序。不过，是有可能通过源码标签这个字段维护一个版本号，并利用它构造一个同 PEP 440 兼容的版本号的。

提示

pbr（即 Python Build Reasonableness，https://pypi.python.org/pypi/pbr）将在 4.2 节中讨论，它可以基于项目的 Git 版本自动生成版本号。

1.4 编码风格与自动检查

没错，编码风格是一个不太讨巧的话题，不过这里仍然要聊一下。

Python 具有其他语言少有的绝佳质量[②]：使用缩进来定义代码块。乍一看，似乎它解决了一个由来已久的“往哪里放大括号？”的问题，然而，它又带来了“如何缩进？”这个新问题。

而 Python 社区则利用他们的无穷智慧，提出了编写 Python 代码的 PEP 8[③]（http://www.python.org/dev/peps/pep-0008/）标准。这些规范可以归纳成下面的内容。

- 每个缩进层级使用 4 个空格。
- 每行最多 79 个字符。
- 顶层的函数或类的定义之间空两行。
- 采用 ASCII 或 UTF-8 编码文件。
- 在文件顶端，注释和文档说明之下，每行每条 `import` 语句只导入一个模块，同时要按标准库、第三方库和本地库的导入顺序进行分组。

① 对于 Git，指的是 `git-describe(1)`。
② 你可能有不同意见。
③ *PEP 8 Style Guide for Python Code*, 5th July 2001, Guido van Rossum, Barry Warsaw, Nick Coghlan

- 在小括号、中括号、大括号之间或者逗号之前没有额外的空格。
- 类的命名采用骆驼命名法，如 `CamelCase`；异常的定义使用 `Error` 前缀（如适用的话）；函数的命名使用小写字符，如 `separated_by_underscores`；用下划线开头定义私有的属性或方法，如 `_private`。

这些规范其实很容易遵守，而且实际上很合理。大部分程序员在按照这些规范写代码时并没有什么不便。

然而，**犯错在所难免**，保持代码符合 PEP 8 规范的要求仍是一件麻烦事。工具 **pep8**（https://pypi.python.org/pypi/pep8）就是用来解决这个问题的，它能自动检查 Python 文件是否符合 PEP 8 要求，如示例 1.1 所示。

示例 1.1　运行 pep8

```
$ pep8 hello.py
hello.py:4:1: E302 expected 2 blank lines, found 1
$ echo $?
1
```

pep8 会显示在哪行哪里违反了 PEP 8，并为每个问题给出其错误码。如果违反了那些必须遵守的规范，则会报出**错误**（以 E 开头的错误码），如果是细小的问题则会报**警告**（以 W 开头的错误码）。跟在字母后面的三位数字则指出具体的错误或警告，可以从错误码的百位数看出问题的大概类别。例如，以 E2 开头的错误通常与空格有关，以 E3 开头的错误则与空行有关，而以 W6 开头的警告则表明使用了已废弃的功能。

社区仍然在争论对并非标准库一部分的代码进行 PEP 8 验证是否是一种好的实践。这里建议还是考虑一下，最好能定期用 PEP 8 验证工具对代码进行检测。一种简单的方式就是将其集成到测试集中。尽管这似乎有点儿极端，但这能保证代码一直遵守 PEP 8 规范。6.7 节中将介绍如何将 pep8 与 tox 集成，从而让这些检查自动化。

OpenStack 项目从一开始就通过自动检查强制遵守 PEP 8 规范。尽管有时候这让新手比较抓狂，但这让整个代码库的每一部分都保持一致，要知道现在它有 167 万行代码。对于任何规模的项目这都是非常重要的，因为即使对于空白的顺序，不同的程序员也会有不同的意见。

也可以使用`--ignore` 选项忽略某些特定的错误或警告，如示例 1.2 所示。

示例 1.2 运行 pep8 时指定--ignore 选项

```
$ pep8 --ignore=E3 hello.py
$ echo $?
0
```

这可以有效地忽略那些不想遵循的 PEP 8 标准。如果使用 pep8 对已有的代码库进行检查，这也可以暂时忽略某些问题，从而每次只专注解决一类问题。

注意

如果正在写一些针对 Python 的 C 语言代码(如模块)，则 PEP 7(http://www.python.org/dev/peps/pep-0007/)标准描述了应该遵循的相应的编码风格。

还有一些其他的工具能够检查真正的编码错误而非风格问题。下面是一些比较知名的工具。

- pyflakes（https://launchpad.net/pyflakes），它支持插件。
- pylint（https://pypi.python.org/pypi/pylint），它支持 PEP 8，默认可以执行更多检查，并且支持插件。

这些工具都是利用静态分析技术，也就是说，解析代码并分析代码而无需运行。

如果选择使用 pyflakes，要注意它按自己的规则检查而非按 PEP 8，所以仍然需要运行 pep8。为了简化操作，一个名为 flake8（https://pypi.python.org/pypi/flake8）的项目将 pyflakes 和 pep8 合并成了一个命令，而且加入了一些新的功能，如忽略带有`#noqa`的行以及通过入口点（entry point）进行扩展。

为了追求优美而统一的代码，OpenStack 选择使用 flake8 进行代码检查。不过随着时间的推移，社区的开发者们已经开始利用 flake8 的可扩展性对提交的代码进行更多潜在问题的检查。最终 flake8 的这个扩展被命名为 hacking（https://pypi.python.org/pypi/hacking）。它可以检查`except`语句的错误使用、Python 2 与 Python 3 的兼容性问题、导入风格、危险的字符串格式化及可能的本地化问题。

如果你正开始一个新项目，这里强烈建议使用上述工具之一对代码的质量和风格进行自动检查。如果已经有了代码库，那么一种比较好的方式是先关闭大部分警告，然后每次只解决一类问题。

尽管没有一种工具能够完美地满足每个项目或者每个人的喜好，但 flake8 和 hacking 的

结合使用是持续改进代码质量的良好方式。要是没想好其他的，那么这是一个向此目标前进的好的开始。

提示

许多文本编辑器，包括流行的 GNU Emacs（http://www.gnu.org/software/emacs/）和 vim（http://www.vim.org/），都有能够直接对代码运行 pep8 和 flake8 这类工具的插件（如 Flymake），能够交互地突出显示代码中任何不兼容 PEP 8 规范的部分。这种方式能够非常方便地在代码编写过程中修正大部分风格错误。

第 2 章 模块和库

2.1 导入系统

要使用模块和库，需要先进行导入。

Python 之禅

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

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

导入系统是相当复杂的，不过你可能已经了解了一些基本知识。这里会介绍一些关于这一子系统的内部机理。

sys 模块包含许多关于 Python 导入系统的信息。首先，当前可导入的模块列表都是通过 sys.moudle 变量才可以使用的。它是一个字典，其中键（key）是模块名字，对应的值（value）是模块对象。

```
>>> sys.modules['os']
<module 'os' from '/usr/lib/python2.7/os.pyc'>
```

许多模块是内置的，这些内置的模块在 sys.builtin_module_names 中列出。内置模块可以根据传入 Python 构建系统的编译选项的不同而变化。

导入模块时，Python 会依赖一个路径列表。这个列表存储在 sys.path 变量中，并且告诉 Python 去哪里搜索要加载的模块。可以在代码中修改这个列表，根据需要添加或删除路径，也可以通过编写 Python 代码直接修改环境变量 PYTHONPATH。下面的方法几乎是相等的[①]。

```
>>> import sys
>>> sys.path.append('/foo/bar')

$ PYTHONPATH=/foo/bar python
>>> import sys
>>> '/foo/bar' in sys.path
True
```

在 sys.path 中顺序很重要，因为需要遍历这个列表来寻找请求的模块。

也可以通过自定义的导入器（importer）对导入机制进行扩展。Hy[②]正是利用的这种技术告诉 Python 如何导入其他非标准的.py 或者.pyc 文件的。

顾名思义，导入钩子机制是由 PEP 302（http://www.python.org/dev/peps/pep-0302/）定义的[③]。它允许扩展标准的导入机制，并对其进行预处理，也可以通过追加一个工厂类到 sys.path_hooks 来添加自定义的模块查找器（finder）。

模块查找器对象必须有一个返回加载器对象的 find_module(fullname, path=None) 方法，这个加载器对象必须包含一个负责从源文件中加载模块的 load_module(fullname)

① 说几乎是因为路径并不会被放在列表的同一级上，尽管根据你的使用情况它可能并不重要。
② Hy 是 Python 上的 Lisp 实现，会在 9.1 节介绍。
③ 在 Python 2.3 版本实现的新的带入钩子机制。