

[笔记][Learning Python][1. 开始]

Python

[笔记][Learning Python][1. 开始]

前言

1. Python问答环节

- 1.1 为什么人们要用Python？
- 1.2 Python是脚本语言吗？
- 1.3 Python的缺点是什么？
- 1.4 如今都有谁在使用Python？
- 1.5 我可以用Python做什么？
- 1.6 Python是怎样被开发和维护的？
- 1.7 Python的技术优势是什么？
- 1.8 Python与其他编程语言比怎么样？
- 1.9 本章总结
- 1.10 检验你的知识：小测试

2. Python是怎样运行程序的

- 2.1 介绍Python解释器
- 2.2 程序执行
- 2.3 执行模型的变体
- 2.4 本章总结
- 2.5 检验你的知识：小测试

3. 你应该如何运行程序

- 3.1 交互提示模式
 - 3.2 系统命令行和文件
 - 3.3 Unix 风格可执行脚本：#!
 - 3.4 点击文件图标
 - 3.5 模块导入和重载
 - 3.6 使用 exec 运行模块文件
 - 3.7 IDLE 用户界面
 - 3.8 其他 IDE
 - 3.9 其他启动方法
 - 3.10 我应该用哪种启动方式？
 - 3.11 本章总结
 - 3.12 检验你的知识：小测试
 - 3.13 检验你的知识：第一部分练习
-

前言

生词表

`raise the bar for newcomers` 提高新手的门槛
`has yet to` 还有待，还没有
`de facto standard` 约定俗成的标准
`granularity` 粒度
`wart` 疣，缺点
`lay groundwork for` 奠定基础

两卷本：

- `Learning Python` (Python 学习手册)
 - 补充知识：`Python Pocket Reference` (Python 袖珍指南)
- `Programming Python` (Python 编程)

本书等价于一个学期长度的 `Python` 课程。

本书同时涵盖 3.3 和 2.7 的内容。

`Python 3.X` 系列是 `Python` 语言的一次断代升级。

`Python 3` 强制使用新式类模型 `new-style class`，将类 `class` 与类型 `type` 合并。

想要从事软件开发行业的读者，在读完本书之后应该还要投入额外的时间，比如继续学习 `Programming Python` 一书。

练习很重要！

本书覆盖了 `Python` 语言本身的知识，不过多涉及它的应用，标准库和第三方工具。

其他重要资源

- 参考资源
 - `Python Pocket Reference`
 - 官方参考手册 <http://www.python.org>
- 应用和库
 - `Programming Python`

快速上手 `Python`

Chapter 1, Chapter 4, Chapter 10, and Chapter 28 (and perhaps 26)

本书源码与习题答案

<http://oreil.ly/LearningPython-5E>

勘误与更新

- 出版社网址(主要是勘误)
<http://oreil.ly/LearningPython-5E>

- 作者网址(主要是更新)
<http://www.rmi.net/~lutz/about-lp5e.html>

1. Python 问答环节

第一章会简要介绍 Python 如此流行的几个主要原因。

1.1 为什么人们要用 Python ?

软件质量

Python 语言易读性 `readability` 高，所以代码可复用、易维护。

开发者生产力

Python 增强开发者生产力，代码量是 C++ 或者 Java 的三分之一到五分之一，所以打的字少，也容易调试，写完之后也容易维护。

程序便携性

适应所有主流电脑平台

库文件支持

Python 标准库非常丰富，还有很多第三方库。

组件集成

Python 可以调用 C 和 C++ 的库，也可以被 C 和 C++ 调用等等。

快乐

因为 Python 的易用性和内置工具箱，使用 Python 编程是一件愉悦的事情。

在 Python 命令交互模式输入 `import this` 来发现彩蛋
得到 Python 的设计原则
其中 EIBTI 是 `explicit is better than implicit` 的缩写

1.2 Python 是脚本语言吗？

Python 通常被定义为 **面向对象的脚本语言** `object-oriented scripting language`
作者认为它应该被认为是 **融合了面向过程、面向对象和函数式编程范式的通用编程语言** `a general-purpose programming language that blends procedural, functional, and object-oriented paradigms`。

人们更喜欢称 Python 的代码文件为 **脚本** 而不是 **程序**，本书会交替使用这两个术语，对于简单的单

文件代码称之为**脚本**，对于复杂的多文件应用称之为**程序**。

脚本这一术语，意味着 `Python` 支持快速和灵活的开发模式，而不是特定的应用领域。

1.3 `Python` 的缺点是什么？

执行速度 `execution speed` 不如一些更底层的语言比如 `C` 和 `C++` 快。

`Python` 的标准实现过程是先把源代码 `source code` 编译 `compile` 成中间代码，即字节码 `byte code`，然后解释执行字节码。

字节码提供了便携性，它是不依赖于平台的格式。

但是由于 `Python` 通常不会直接编译成二进制机器码（例如对英特尔芯片的指令），一些程序在 `Python` 中比起那些完全编译的语言（比如 `C` 语言）跑的更慢。

`PyPy` 系统可以在执行程序的时候进一步编译代码，所以可以加速到10倍到100倍，是另外一种可选实现。

`Python` 的快速开发能力很多时候比较慢的执行速度更重要。

你可以把需要快速运算的代码分割开来，做成已编译扩展 `compiled extensions`，让 `Python` 调用它们。

`NumPy` 是把 `Python` 用作控制语言的一个很好的例子。

其他缺点

`Python` 语言和它的标准库更新速度快。

1.4 如今都有谁在使用 `Python` ？

如今大约有100万的 `Python` 使用者 `1 million Python users`

大型公司：

- `Google` 谷歌
- `YouTube` 油管
- `Dropbox`
- `Raspberri Pi` 树莓派
- `EVE Online`
- `BitTorrent`
- 动画公司
- `ESRI`
- `Google's App Engine` `web` 开发框架
- `IronPort` 电子邮件服务器，使用了100万行以上 `Python` 代码来完成任务
- `Maya` 使用 `Python` 作为脚本
- `NSA` 美国国家安全局
- `iRobot`
- `Civilization IV` 脚本
- `One Laptop Per Child (OLPC) project`

- `Netflix` and `Yelp`
- `Intel`, `Cisco`, `Hewlett-Packard`, `Seagate`, `Qualcomm`, and `IBM` 使用 `Python` 做硬件测试
- `JPMorgan Chase`, `UBS`, `Getco`, and `Citadel` 用 `Python` 预测金融市场
- `NASA`, `Los Alamos`, `Fermilab`, `JPL` 用 `Python` 做科研

更多详见：

- Success stories: <http://www.python.org/about/success>
- Application domains: <http://www.python.org/about/apps>
- User quotes: <http://www.python.org/about/quotes>
- Wikipedia page: http://en.wikipedia.org/wiki/List_of_Python_software

1.5 我可以用 `Python` 做什么？

- 系统编程：搜索文件，启动其他程序，多线程多进程工作
- GUI 编程：`tkinter` 等等
- 网络编程
- 组件集成：可扩展（可被 `C` 和 `C++` 扩展），可嵌入（可以嵌入到 `C` 和 `C++`）
- 数据库编程：`Object Relational Mappers`（`ORM`）
- 快速原型：一开始用 `Python`，然后用 `C` 或者 `C++` 编译需要运算速度的部分，之后再发布
- 数学和科学编程：`NumPy`，`Python` 的 `PyPy` 实现，在科学计算领域的代码执行速度可以提高到10倍到100倍
- 更多：游戏，图像，数据挖掘，机器人，`Excel`

1.6 `Python` 是怎样被开发和维护的？

PEP（`Python Enhancement Proposal`）

PSF（`Python Software Foundation`）

会议：`O'Reilly's OSCON` 和 `PSF's PyCon`

1.7 `Python` 的技术优势是什么？

- 支持面向对象、面向过程、函数式编程，而这些都是可选的
- 免费
- 便携性

`Python` 创始人 Guido van Rossum

- 强大：动态类型、自动内存分配、支持大型项目编程、内建对象类型、内建工具、丰富的库功能、第三方工具
 - 可混合：带来了快速原型
 - 易用性：有人叫 `Python` 可执行的伪代码 `executable pseudocode`
 - 容易学
 - 命名来自于 `Monty Python`：变量名 `foo` 和 `bar` 会被 `spam` 和 `eggs` 所替代
-

1.8 `Python` 与其他编程语言比怎么样？

- 比 `Tcl` 更强大，适用于大型系统
 - 比 `Perl` 更易读
 - 比 `Java` 和 `C#` 更简单和易用
 - 比 `C++` 更简单和易用
 - 比 `C` 更高级，更简单
 - 比 `Visual Basic` 更强大，用途广和跨平台
 - 比 `PHP` 更易读，用途更广
 - 比 `JavaScript` 更强大，用途更广
 - 比 `Ruby` 更易读
 - 比 `Lua` 更成熟，适用性更好
 - 比 `Smalltalk`，`Lisp` 和 `Prolog` 更简单
-

1.9 本章总结

1.10 检验你的知识：小测试

2. `Python` 是怎样运行程序的

2.1 介绍 `Python` 解释器

解释器 `interpreter` 是运行其他程序的程序，是位于你的代码和电脑硬件之间的软件逻辑层。解释器本身可以被多种语言实现，比如 `C` 语言，`Java` 语言等等。

不同平台安装 `Python` 的过程，见附录 A

Python 可以在官网下载：

<http://www.python.org>

Unix 和 Linux 系统，Python 很可能在 /usr 目录树下面

2.2 程序执行

程序员视角

通常情况下，Python 程序文件以 .py 结尾，实际上，只有那些需要导入的模块才需要加上 .py 扩展名，但是为了一致性，大多数 Python 文件都以 .py 结尾。

执行 execute 文件，意味着从顶至底，一一运行所有命令。

在 Windows 命令行模式下，python script0.py

Python 视角

字节码 byte code 编译

- 首先把源代码 source code 编译成字节码 byte code
 - 字节码是更底层，平台无关的代码
 - 编译成字节码是因为它们运行的更快
 - 这一过程对你几乎是隐藏的
 - 如果有写入权限，Python 会在你的电脑上存储 .pyc 文件，即 compiled ".py" source
 - 3.2 之前，Python 会在相同位置保存字节码，比如导入 script.py 之后会在相同路径出现 script.pyc
 - 3.2 及以后，把字节码保存在 __pycache__ 里面，并且会标明创建该文件的 Python 版本，比如 script.cpython-33.pyc
 - 只要你没有修改源代码，或者运行一个不同版本的 Python，Python 会加载 .pyc 文件，并跳过编译来加速启动。
 - 源代码改变：Python 会自动检查上一次修改时的时间戳 time stamp 和字节码文件，如果被修改，在下一次运行程序之前，字节码会被重新编译。
 - Python 版本变化：import 会检查文件是否是被另外的 Python 版本所创建，如果是则要重新编译。Python 3.2 以前，用的是字节码里的魔法版本号 magic version number，3.2 及以后，用的是字节码的文件名。
 - 当 Python 没有写入权限时，字节码会被编译，然后保存在内存里，退出的时候就会丢弃。
 - 写大型程序的时候最好保证 Python 有写入权限，因为可以更快启动。
 - 字节码也是发布 Python 的一个方法，如果完全没有 .py 文件，Python 也可以依靠 .pyc 文件执行。
 - 只有导入 import 的文件才会以文件的形式保存它的字节码。
-

Python 虚拟机

- Python Virtual Machine，简称 PVM

- 编译成字节码之后，送到 **PVM** 那里去执行

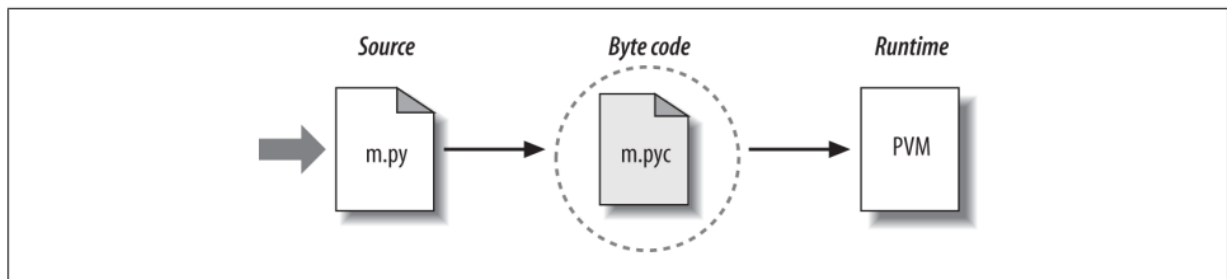


Figure 2-2. Python's traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

2.3 执行模型的变体

Python 语言目前的五种实现：

- **CPython**：标准实现，大多数读者都会使用
- **Jython**：直接访问 **Java** 组件。图 2.2 的右边两项变成 **Java** 字节码和 **Java** 虚拟机 **JVM**。可以导入并使用 **Java** 类。官网：<http://jython.org>
- **IronPython**：直接访问 **.NET** 组件。官网：<http://ironpython.net>
- **Stackless**：注重于并发 **concurrency** 编程。**EVE Online** 使用了 **Stackless Python**。官网：<http://stackless.com>
- **PyPy**：注重于性能 **performance**。是 **CPython** 的替代，执行大多数代码都比 **CPython** 快。它的 **JIT (just in time)** 编译器是 **PVM** 的一个扩展，可以直接将一部分字节码翻译成二进制机器码。目前它的速度是 **CPython** 的 5.7 倍（见<http://speed.pypy.org/>）。有时候它占用的内存也比 **CPython** 要少。官网：<http://pypy.org>。当前的性能详见<http://www.pypy.org/performance.html>。如果你需要写 **CPU** 密集型的代码（**CPU-intensive code**），**PyPy** 值得关注。

另外还有一些非标准的 **Python** 实现：

- **Cython**：**Python** 与 **C** 混合，官网：<http://cython.org>
- **Shed Skin**：隐式静态类型。把 **Python** 代码翻译成 **C++** 代码，然后让系统里的 **C++** 编译器编译成机器码。

冻结二进制文件 **Frozen Binaries**

冻结二进制文件 **frozen binaries** 不需要安装 **Python** 即可运行。

冻结二进制文件把你的**字节码文件**、**解释器 PVM** 和其他必需的 **Python 支持文件** 打包，最终形成一个二进制可执行文件 **.exe**。

- **py2exe**：仅支持 **Windows** 平台
- **PyInstaller**：支持主流平台，并且可以生成可安装文件
- **py2app**：**Mac OS X**
- **cx_freeze**：支持 **Python 3.X**，跨平台

冻结二进制文件的输出跟一个真正的编译器还是有差别的，它们仍然通过虚拟机来执行字节码，所以它们的执行速度跟源代码文件差不多。

通常冻结二进制文件的体积不会小，然而也不会过大。

在运行冻结二进制文件的机器上不用安装 Python，并且你的代码得到一定的隐藏。

未来的可能性

- 可能会出现直接把 Python 代码编译成机器码的编译器
 - 新的字节码格式（比如 Parrot 项目 <http://parrot.org>）
 - 由于 Python 高度的动态特性，未来的 Python 实现很可能会保留当前的 PVM
-

2.4 本章总结

本章介绍了 Python 的运行模型：Python 是如何执行你的程序的。

探索了该模型的一些变体。

2.5 检验你的知识：小测试

1. 什么是 Python 解释器？

Python 解释器是执行你写的程序的程序。

2. 什么是源代码？

源代码是你为你的程序写的一系列语句。它以文本的形式存在于文本文件中，通常以 .py 为扩展名。

3. 什么是字节码？

是 Python 编译后的较低层的代码。Python 自动将字节码用 .pyc 扩展名保存起来。

4. 什么是 PVM？

Python Virtual Machine，即 Python 虚拟机，是 Python 的运行引擎，用来解释你编译好的字节码。

5. 说出两个或更多的 Python 标准运行模型的变体。

Psyco，Shed Skin 和冻结二进制文件都是 Python 标准运行模型的变体。

6. CPython，JPython 和 IronPython 有什么区别？

CPython 是 Python 语言的标准实现。

JPython 和 IronPython 让 Python 程序可以分别应用于 Java 和 .NET 环境。

它们是 Python 的可选编译器。

7. 什么是 Stackless 和 PyPy？

Stackless 是专注于并发的 Python 增强版本。

PyPy 是 Python 的重新实现，专注于速度。它是 Psyco 的继任者，继承了由 Psyco 发起的 JIT 概念。

3. 你应该如何运行程序

目标：学习如何运行 Python 代码。

3.1 交互提示模式

开启一个交互会话

`interactive command line` 交互命令行

`interactive prompt` 交互提示模式

在系统提示模式下，输入 `python`，不带任何参数，进入 Python 的交互提示模式

本书中 % 代表通用的系统提示符

退出：Windows 下用 `Ctrl-Z`，Unix 下用 `Ctrl-D`

如何进入系统 shell 提示符 (`system shell prompt`)

- Windows：在 `cmd.exe` (也叫命令提示符 `Command Prompt`) 里面输入 `python`
- Mac OS X：双击 `Applications - Utilities - Terminal`，输入 `python`
- Linux：在 `shell` 或者 `terminal` 里输入 `python`

不敲命令进入 Python 交互提示符模式

- Windows 7 及之前版本：启动 `IDLE GUI` 或者在开始菜单选择 `Python (command line)`
- Windows 8：没有开始按钮，查看附录 A

看到 `>>>` 说明你进入了 Python 的交互解释会话，你可以输入语句 `statement` 和表达式 `expression`，并立即执行它。

Windows 的命令提示符 `Command Prompt` 在哪？

【!】略

提到了 `Cygwin` 系统

系统路径 `System Path`

需要在系统的 `PATH` 环境变量中添加 Python 的安装目录，才能直接在命令提示符模式输入 `python` 进入交互提示模式。

否则需要输入 Python 可执行文件的完整路径，在 Unix 和 Linux 系统，输入类

似 `/usr/local/bin/python` 或者 `/usr/bin/python3`

在 Windows 系统，输入 `C:\Python33\python`

你还可以先使用 `cd` 命令进入 Python 安装目录，然后直接输入 `python`

但是最终你可能还是想设置 `PATH`，这样就可以直接输入 `python`，参见附录 A

3.3 中的新 Windows 选项

- 从 3.3 开始，安装 Python 时可以勾选自动把 Python 3.3 的目录加入到系统 PATH 中。注意在安装时它默认是取消勾选的。
- Python 3.3 安装新的 Windows 启动器，这是两个新的可执行程序，py 带控制台，pyw 无控制台，它们直接安装在系统目录下，不用配置 PATH 环境变量即可使用。

```
> Windows PowerShell
PS C:\Users\jpch8> py
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ^Z

PS C:\Users\jpch8> py -2
Requested Python version (2) not installed
PS C:\Users\jpch8> py -3.1
Requested Python version (3.1) not installed
PS C:\Users\jpch8> _
```

py 后面还可以指定版本号。

关于这个启动器，详见附录 B。

基本来说，本书所有代码都可以用 py 来代替 python，它可以让你控制运行代码的 Python 版本号。

从哪里运行：代码文件夹

本书会从 C:\code 文件夹来运行代码。

如何新建文件夹：

- Windows 平台上，先 cd C:\ 然后再 mkdir code；或者在文件浏览器中右键，新建文件夹。
- 基于 Unix 的平台，可以在 /usr/home 路径下 mkdir，或者通过 GUI 创建文件夹。

什么时候不要打字：提示符和注释

系统提示符用 % 后面加一个空格表示，在 Windows 下还会用 C:\code> 加一个空格表示。

不要打这些符号！

同样也不要打出现在行首的 >>> 和 ...，这是 Python 交互模式的提示符。

类似的，也不用打 # 和后面的东西，通常它是注释，除非是用在 Unix 系统上（后面和附录 B 会讲）。

以交互式运行代码

当以交互模式工作的时候，代码的结果会在你按下回车键之后，展现在 >>> 输入行下面。

```
% python
>>> print('Hello world!')
Hello world!
>>> print(2 ** 8)
256
```

注意 `Python 2` 的 `print` 不需要括号，而在 `Python 3` 里面，`print` 是一个函数，需要括号调用。

`2 ** 8` 是 `2` 的 `8` 次方。

在交互模式中自动输出表达式的结果，所以不用显式的调用 `print`。

```
>>> lumberjack = 'okay'
>>> lumberjack
'okay'
>>> 2 ** 8
256
>>> ^Z # Use Ctrl-D (on Unix) or Ctrl-Z (on Windows) to exit
%
```

变量 `variable`

表达式 `expression`

`#` 注释，不需要打印，在系统提示符会出错

你也可以运行多条语句 `statement`，当你输入完所有的语句之后，按下空格两次，插入就会立即执行。

为什么选择交互提示模式？

因为你可以方便的试验 `experiment` 和测试 `test` 代码。

试验

如果你对一段 `Python` 代码是如何工作的还有疑问，那就打开交互式命令行试一试。

比如可以尝试下面的代码：

```
% python
>>> 'Spam!' * 8 # Learning by trying
'Spam! Spam! Spam! Spam! Spam! Spam! Spam!'
```

字符串重复 `string repetition`

在 `Python` 中，`*` 对于数字意味着乘法，对于字符串意味着重复。

而且出错了也不会有什么麻烦：

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

在 `Python` 中，赋值之前使用一个变量会报错。

测试

交互解释器也是测试你写在文件中的代码的好工具。

你可以先导入模块文件然后使用。

```
>>> import os
>>> os.getcwd() # Testing on the fly
'c:\\code'
```

使用说明：交互提示模式

- 只能输入 `Python` 命令。输入系统命令要通过其他方式，比如 `os.system` 模块。
- `print` 只有在文件里面才是必需的。因为交互解释器会自动打印表达式的结果。
- 注意输入复合语句时提示符的变化。使用回车或者 `Ctrl-C` 组合来回到主提示符。 `>>>` 和 `...` 提示符也可以在内置模块 `sys` 中改变。
- 用一行空白行结束复合语句。
- 交互提示符每次只执行一个语句。

输入多行（复合）语句

- 第一，记住用空行结束复合语句（比如 `for` 循环和 `if` 条件测试语句）。 `...` 续航提示符是 `Python` 自动打印的，自己不要输入。
- 第二，交互提示符每次只运行一条语句。

```
>>> for x in 'spam':
...     print(x) # Press Enter twice before a new statement
...     print('done')
File "<stdin>", line 3
print('done')
^
SyntaxError: invalid syntax
```

3.2 系统命令行和文件

为了永久保存程序，你需要把代码写入文件，通常被称作模块 `modules`。

模块是包含 `Python` 语句的文件。

在 `Python` 中，模块文件也经常被称为程序 `programs`。

可以直接运行的模块文件有时也叫做脚本 `scripts`，这是顶层程序文件 `top-level program file` 的不正式的讲法。

模块 `module` 是从另外的文件中导入的文件；而脚本 `script` 是程序的主文件。

对于很多程序员，一个系统 `shell` 命令行窗口，加上一个文本编辑器窗口，就组成了他们所需要的集成开发环境 `integrated development environment`。

第一个脚本

script1.py

```
# A first Python script
import sys # Load a library module
print(sys.platform)
print(2 ** 100) # Raise 2 to a power
x = 'Spam!'
print(x * 8) # String repetition
```

`sys.platform` 是说明你电脑平台的字符串

`#` 后面的字符是 `Python` 的注释。注释可以在一行的右边，也可以独自成行。

只能导入 `import` 以 `.py` 结尾的文件。

因为你可能想要在将来导入你写的程序，所以最好对你写的绝大部分 `Python` 程序都用 `.py` 后缀名。

如果后缀名不存在的话，你很可能享受不到语法着色 `syntax colorization` 和自动缩进。

用命令行运行文件

可以在系统 `shell` 提示符运行文件。

用 `Python` 命令加上文件全名。

```
% python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

命令行用法的变体

把输出保存到文件：

```
% python script1.py > saveit.txt
```

通常被称为流重定向 `stream redirection`。

如果没有配置环境变量，需要这样执行程序：

```
C:\code> C:\python33\python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

也可以使用 `Python 3.3` 支持的 `Windows` 启动器，你还可以指定 `Python` 版本：

```
c:\code> py -3 script1.py
win32
1267650600228229401496703205376
```

```
Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

在 `Windows` 里面可以省略 `python` , 直接输入文件名 :

```
C:\code> script1.py
```

原因是 `Windows` 注册表 (`Windows Registry`) , 也叫文件名关联 (`filename association`) 可以找到用什么程序执行一个文件。

如果文件不在当前工作目录, 记得要给出文件的绝对路径 :

```
C:\code> cd D:\other
D:\other> python c:\code\script1.py
```

如果 `PATH` 环境变量没有包括 `Python` 的目录, 而且脚本文件也不在当前工作目录 :

```
D:\other> C:\Python33\python c:\code\script1.py
```

使用说明 : 命令行和文件

- 注意 `Windows` 和 `IDLE` 的自动扩展名。
- 在系统提示符中使用文件扩展名和目录路径, 但是不要用在 `import` 语句中。
- 在文件中使用 `print` 语句 `statement` 。在交互提示符中, `print` 往往是多余的。

3.3 Unix 风格可执行脚本 : `#!`

Unix 脚本基础

如果你想在类 `Unix` 系统中使用 `Python` , 你可以把 `Python` 代码文件编程可执行的程序, 这种文件被叫做可执行脚本 `executable scripts` 。

`Unix` 风格的可执行脚本就是包含 `Python` 语句的普通文本文件, 但是它具有两种特性 :

- 第一行是特殊的。以 `#!` 开头 (被称作 `hash bang` 或者 `shebang`) , 后面跟着 `Python` 解释器的路径。
- 它们通常有可执行权限。在 `Unix` 系统中, 使用 `chmod +x file.py` 即可。

例如编写一个 `brian` 文件 :

```
#!/usr/local/bin/python
print('The Bright Side ' + 'of Life...') # + means concatenate for string
s
```

实际上第一行是 `Python` 注释, 但是在 `Unix` 系统上它是特殊的, 因为操作系统用它来为文件找到解释器。

`chmod +x brian` 然后直接运行 `brian` 即可执行。
注意这个文件没有 `.py` 后缀名就能执行，虽然加上也没关系。

Unix env 查找技巧

在一些 Unix 系统里，你可以不用把到 Python 解释器的路径写死。

```
#!/usr/bin/env python
...script goes here...
```

这样写的话，`env` 程序会根据你的系统搜索路径设置来定位到 Python 解释器（在大多数 Unix 系统中，是遍历 `PATH` 环境变量里的目录）。
这种方式具有更强的便携性，所以推荐使用。

Python 3.3 的 Windows 启动器：#! 来到 Windows

`python brian` 即可执行，不需要 `#!`，文件也不需要可执行权限。

在 Python 3.3 及以后，Windows 启动器除了带来 `py` 可执行程序，它还会解析 `#!` 行来确定用哪个版本的 Python 来运行你的代码。

你可以用完整或者部分的版本号来指定。

启动器的 `#!` 解析机制会在两种情况下生效：

- 命令行下使用 `py` 来运行代码
- 双击运行代码（`py` 通过文件名关联被隐式调用）

比如：

```
c:\code> type robin3.py
#!/usr/bin/python3
print('Run', 'away!...') # 3.X function
c:\code> py robin3.py # Run file per #! line version
Run away!...
c:\code> type robin2.py
#!/python2
print 'Run', 'away more!...' # 2.X statement
c:\code> py robin2.py # Run file per #! line version
Run away more!...
```

在命令行中传递版本号同样奏效（跟启动交互提示模式一样）：

```
c:\code> py -3.1 robin3.py # Run per command-line argument
Run away!...
```

3.4 点击文件图标

Windows 的图标点击：

Python 安装的时候，使用 Windows 的文件名关联，把自己注册为打开 Python 代码的程序。在 Python 3.3 以前，Python 使用 `python.exe` 和 `pythonw.exe` 来分别打开 `.py` 和 `.pyw` 文件；在 Python 3.3 及以后，Python 使用 `py.exe` 和 `pyw.exe` 来分别打开 `.py` 和 `.pyw` 文件。

非 Windows 系统的图标点击：

【!】略

在 Windows 上点击图标

- 源文件是白底的。
- 字节码是黑底的。

一般来说你要点击白底的源文件，那里有代码的最新更改。

Windows 的 input 技巧

运行文件之后控制台一闪而过，只要在文件末尾加一句 `input()` 即可（对于 3.X 版本），对于 2.X 版本，使用 `raw_input`。

它的实际效果是暂停脚本，等待输入。

这个技巧仅仅适用于以下情况：

- Windows 平台
- 以双击的方式运行程序
- 你的脚本只打印一些输入随即退出

只有当上述三种情况同时存在时，才需要加一句 `input()`。

```
Python 2 的 raw_input() 被重命名为 Python 3 的 input() 。
Python 2 的 input() 会对输入的字符串进行运算。
Python 3 中可以通过 eval(input()) 来模拟 Python 2 的 input() 。
```

另外，以 `.pyw` 结尾的文件不会弹出控制台。

图标点击执行的其他限制

- 看不到 Python 输出的错误信息
- 就算在最后添加 `input()` 也可能没用，因为在到达 `input()` 之前，程序就出错跳出了
- 可以用异常处理来处理错误，也可以将错误信息重定向到文件

推荐使用系统命令行或者 IDLE 来运行程序，点击图标运行最好在 debug 之后再尝试。

3.5 模块导入和重载

导入也是运行程序的一种方式。

导入和重载基础

每个以 `.py` 结尾的 `Python` 源代码文件都是一个模块。

其他文件可以通过导入操作来访问一个模块的内容。

导入操作：加载另一个文件，并得到授权可以访问该文件的内容。

一个模块的内容通过**属性**开放给外部。

基于模块的服务是 `Python` 的程序架构 `program architecture` 的核心理念。

有一个模块做为主文件或者顶层文件（`main or top-level file`），或者说脚本 `script`，它启动整个程序，按行顺序执行。除此之外，都是模块在导入模块。

导入操作的最后一个步骤是运行被导入模块的代码。

所以导入也是另外一种运行程序的方法。

以 `script1.py` 为例，第一次导入可以运行，此后的导入都不行。

即便你修改了 `script1.py` 的源码并保存，再导入，也不行。

这是被故意设计成这样的，因为导入操作开销太大，需要找到文件，编译成字节码，然后运行文件。

在不停止或者重启会话的情况下，如果你真的想在同一个会话（实际上是进程）中重新运行文件，你需要使用来自 `imp` 标准库中的 `reload` 函数。

```
>>> from imp import reload # Must load from module in 3.X (only)
>>> reload(script1)
win32
65536
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
<module 'script1' from '.*\script1.py'>
>>>
```

在 `Python 2.X` 中，`reload()` 是内置函数，不用导入
`Python 3.X` 必须 `from imp import reload`

`from` 语句从一个模块中拷贝了一个名字。

`reload` 函数加载并运行当前的代码文件。

- `reload()` 接收一个已经被成功导入 `import` 的模块
- `reload()` 有括号，是一个被调用的函数；`import` 没括号，是一个语句。

`reload` is a function that is called, and `import` is a statement.

所以必须把模块名字当作参数传递给 `reload()`，最后多了一行输出是 `reload()` 的返回值（一个模块对象）的展现。

在 `Python 3.X` 中，`reload()` 有两种调用方式：

```
from imp import reload
reload(模块名)
```

或者：

```
import imp
imp.reload(模块名)
```

另外，通过 `from` 加载的名字不会直接被 `reload` 更新；而通过 `import` 加载的名字会立即更新。

如果发现 `reload` 之后名字没有更新，尝试用 `import` 和 `module.attribute` 来引用。

模块的故事：属性

导入操作的最后一步是执行文件，因此 `import` 和 `reload` 提供了另外一种启动程序的选择。

更多的时候，模块一般扮演着工具库 `libraries of tools` 的角色。

模块基本上是变量名字的包，被称作命名空间 `namespace`，包里面的名字被称作属性 `attributes`。

属性是贴在一个具体对象（比如模块）上的变量名字。

模块文件的名字有三种获取方式：

- `from`
- `import`
- `reload`

建立一个最简单的模块 `myfile.py`，内容如下：

```
title = "The Meaning of Life"
```

当这个模块被导入时，它的代码被执行，产生该模块的属性。

即赋值语句创建了一个变量和模块属性，名字叫做 `title`。

```
% python # Start Python
>>> import myfile # Run file; load module as a whole
>>> myfile.title # Use its attribute names: '.' to qualify
'The Meaning of Life'
```

点号 `.` 语法 `object.attribute` 让你可以获取贴在任何对象上的任何属性。
点号可以修饰、限定 `qualify` 模块名。

同样你可以用 `from` 语句从模块里面拷贝名字：

```
% python # Start Python
>>> from myfile import title # Run file; copy its names
>>> title # Use name directly: no need to qualify
'The Meaning of Life'
```

`from` 拷贝一个模块的属性，这样一来它们就成为导入者的变量。如本例中，你可以像变量一样使用被导入的字符串 `title`，而不用使用属性引用（`an attribute reference`）的方式 `myfile.title`。

不管是 `import` 还是 `from`，都不用写 `.py` 扩展名。
在第 5 章会学到，当 Python 寻找文件的时候，知道加入文件后缀 `suffix`。
但是在命令行中必须包括扩展名。

通常模块里会定义多个名字供内部或者外部使用，比如 `threenames.py`：

```
a = 'dead' # Define three attributes
b = 'parrot' # Exported to other files
c = 'sketch'
print(a, b, c) # Also used in this file (in 2.X: print a, b, c)
```

在 Python 2 里面 `print` 要去掉括号 `print a, b, c`。

使用 `import` 得到带属性的模块，使用 `from` 得到文件内名字的拷贝。

当作为顶层文件运行的时候，`threenames.py` 文件可以使用自己的变量，比如：

```
% python threenames.py
dead parrot sketch
```

当该文件第一次被导入的时候，文件的所有代码照常运行一遍，并且使用 `import` 语句的获得一个带属性的模块，使用 `from` 语句的会获得一份该文件的名字拷贝。

```
% python
>>> import threenames # Grab the whole module: it runs here
dead parrot sketch
>>>
>>> threenames.b, threenames.c # Access its attributes
('parrot', 'sketch')
>>>
>>> from threenames import a, b, c # Copy multiple names out
>>> b, c
```

```
('parrot', 'sketch')
```

打印输出的结果带括号，因为它们实际上是元组 `tuple`。

内置函数 `dir` 可以获取模块所有可用名字列表。

```
>>> dir(threenames)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'a',
'b', 'c']
```

这个列表的内容根据 `Python` 版本不同会发生变化。

模块和命名空间

模块同样也是 `Python` 最大的语言结构。

`Python` 程序由通过 `import` 语句联系起来的多个模块文件组成，每个模块文件都是变量的包——即命名空间。

每个模块都是独立的命名空间 `self-contained namespace`：一个模块无法看到定义在其他模块里的名字除非显式导入它。

因此模块起到了减少命名冲突 `minimize name collisions` 的作用，一个文件里的名字不会跟其他文件里的名字冲突，即便它们拼写完全相同。

目前为止，模块有两个优点：

- 代码复用
- 减少命名冲突

`from` 语句实际上让命名空间的划分失效。

假如出现重名，被导入文件的名称会覆盖导入文件的名称。

但是 `from` 语句可以减少打字次数，并且导入的名称是可控的。就像在使用赋值语句一样。

用法说明：`import` 和 `reload`

有些时候，人们会忘记其他运行文件的方法，而一直使用 `import` 和 `reload`。这种方法使用起来很容易让人困惑：

- `reload` 的时候你要确定你已经 `import` 了
- 你要记住 `reload` 要带括号，而 `import` 不用
- 只有 `reload` 才能运行最新修改的代码
- `reload` 不具有传递性，所以有时候你需要 `reload` 多个文件

暂时不用这种技巧，而使用 `IDLE` 的 `Run -- Run Module` 或者命令行模式运行是比较好的。

另外如果你要导入不在当前工作目录的模块时，你需要看第 22 章，学习模块搜索路径 `module search path`。

所以暂时把所有你需要的文件都放在当前工作目录。

但是 `import` 和 `reload` 是非常流行的 Python 类的测试技巧。

Python 会在 `sys.path` 列出的所有目录中搜索模块文件。
`sys.path` 是 `sys` 模块里的一个列表，该列表是由目录路径的字符串组成。
`sys.path` 由 `PYTHONPATH` 环境变量生成，加上一系列的标准目录。
如果你想导入非当前工作目录的文件，需要把被导入文件的目录加入到 `PYTHONPATH` 设置。
详情见第 22 章和附录 A。

3.6 使用 `exec` 运行模块文件

在交互提示模式下，使用 `exec(open('module.py').read())` 来运行模块文件，不用 `import` 也不用 `reload`。

```
% python
>>> exec(open('script1.py').read())
win32
65536
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
...Change script1.py in a text edit window to print 2 ** 32...
>>> exec(open('script1.py').read())
```

使用 `exec` 就像把该文件粘贴到 `exec` 语句被调用的地方，然后运行。
缺点是跟 `from` 一样，有可能会静默地覆盖变量。

```
>>> x = 999
>>> exec(open('script1.py').read()) # Code run in this namespace by default
...same output...
>>> x # Its assignments can overwrite names here
'Spam!'
```

`import` 语句在一次进程 `process` 中，只能运行被导入文件一次，但是不会产生命名冲突，代价是改变代码之后需要 `reload`。

Python 2.X 中，可以使用内置函数 `execfile('module.py')` 或者 `exec(open('module.py').read())`
Python 3.X 只能用 `exec(open('module.py').read())`，注意这个命令在 Python 2.X 中也是可用的。

因为 Python 3.X 的 `exec` 运行模块文件太难打了，建议使用更简单的方式，比如命令行模式。

3.7 IDLE 用户界面

IDLE 通常被称为集成开发环境 IDE (`integrated development environment`)

IDLE 使用标准库 `tkinter GUI toolkit` 搭建窗体。

IDLE 启动细节

各平台启动方法不一样。

但是一般都可以用命令行模式启动：

```
c:\code> python -m idlelib.idle # Run idle.py in a package on module path
```

附录 A 查看 `-m` 标识符的用法，第 V 部分查看 `.` 点号的用法。

IDLE 基础用法

【!】省略

IDLE 用法特性

上一条命令：`Alt-P`

下一条命令：`Alt-N`

在一些 Mac 系统上，试着用 `Ctrl-P` 或者 `Ctrl-N`

你还可以放置光标在命令上，点击，然后回车粘贴。

或者使用标准的复制粘贴操作。

除了命令历史和语法着色 `syntax colorization`，IDLE 还有其他的易用功能，比如：

- 自动缩进
- 自动补全（用 `Tab` 键）
- 调用函数时的气泡提示（键入左括号时）
- 对象属性的选择列表（键入点号时）

高级 IDLE 工具

图形化 `debugger` 和 对象浏览器 `object browser`

`Debug - Debugger` 启用之后可以右键单击，设置断点

另外右键单击出错信息，可以跳转到出错的行数，便于快速修复。

对象浏览器在 `File` 下拉菜单中。

用法说明：IDLE

- 保存文件时必须加 `.py` 后缀名
- 使用 `Run - Run Module` 来运行文件

- 在你需要交互式的测试文件时，使用 `reload`，另外可以用 `Alt-P` 和 `Alt-N` 找到之前的命令
- 你可以定制化 `IDLE`
- `IDLE` 目前没有清屏：你可以敲很多回车，或者打印一堆空行
- `tkinter` GUI 和多线程程序可能无法在 `IDLE` 中正常工作。有时即使在你的代码中使用 `quit` 来退出 GUI 程序，也会导致 `IDLE` 挂起 `hang`。可能使用 `destroy` 会好一些。
- 假如碰到连接错误，使用单进程模式启动 `IDLE`，不要启动用户代码子进程。在 `Windows` 中，在 `C:\Python33\Lib\idlelib` 下面用命令行 `idle.py -n` 或者 `python -m idlelib.idle -n`。
- 小心 `IDLE` 的一些易用特性，这些特性是 `IDLE` 特有的，而非 `Python` 特有，所以出了 `IDLE` 行不通。比如 `IDLE` 在它自己的交互命名空间运行你的脚本，所以你的代码中的变量会在 `IDLE` 交互式会话中自动出现，你不用总是 `import`。另外当你运行一个代码文件时，`IDLE` 会自动变更到该代码文件的目录，并把该目录加入到模块导入搜索路径，你就不用设置搜索路径了。但是这些特性走出 `IDLE` 环境都是没有的。

3.8 其他 IDE

`Eclipse` 和 `PyDev`：安装 `PyDev` 插件之后，`Eclipse` 同样支持 `Python` 开发

`Komodo`：不免费，<http://www.activestate.com>。

`NetBeans IDE for Python`：它还可以开发 `CPython` 和 `JPython` 代码。

`PythonWin`：免费的 `Windows` 平台 IDE，作为 `ActivePython` 发行版中的一部分发行。它支持 `COM` 对象。今天的 `IDLE` 在一些地方比 `PythonWin` 先进，但是仍然有一些工具 `PythonWin` 提供了而 `IDLE` 没有。<http://www.activestate.com>。

`Wing`, `Visual Studio` 和其他：其他商业 IDE 包括 `Wing IDE`，`Microsoft Visual Studio` 配合插件，`PyCharm`，`PyScripter`，`Pyshield` 和 `Spyder`。还有 `Emacs` 和 `Vim`。

3.9 其他启动方法

嵌入式调用

`Python` 程序嵌入 `embedded in` 在另一个程序中（即被另一个程序调用 `run by`）。

用户可以通过 `Python` 来改变系统，因为 `Python` 代码是解释执行的，所以整个系统不用重新编译。

举个例子：

```
#include <Python.h>
Py_Initialize(); // This is C, not Python
PyRun_SimpleString("x = 'brave ' + 'sir robin'"); // But it runs Python code
```

`C` 程序通过连接 `Python` 库来嵌入了 `Python` 解释器，然后传入了一个简单的 `Python` 赋值语句来运行。

具体内容可以找一本 `Python / C 集成 integration` 的书来看。

比如 `Programming Python` 里面讲了在 `C / C++` 中嵌入 `Python`，嵌入 `API` 可以直接调用 `Python` 函数，加载模块等等。而 `Jython` 系统让 `Java` 程序可以通过基于 `Java` 的 `API`（一个 `Python` 解释器类）来调用 `Python` 代码。

冻结二进制可执行文件

冻结二进制可执行文件，是由你程序的字节码以及 `Python` 解释器程序的组合成的包，最后形成一个单一可执行文件。

通常开发过程中不会这样做，只会在发布程序之前打包。

文本编辑器的运行选项

很多文本编辑器也可以运行 `Python` 程序。

其他运行选项

不同的平台可能会有独特的运行方式，比如 `Macintosh` 系统可以拖拽程序到 `Python` 解释器来运行。`Windows` 平台可以通过开始菜单的运行来运行 `Python` 脚本。另外，`Python` 标准库允许 `Python` 程序被其他 `Python` 程序在不同的进程中调用，比如 `os.popen` 和 `os.system` 模块。还有一个 `Web` 页面可能会调用服务器上的脚本。

未来的可能性

【!】略

3.10 我应该用哪种启动方式？

初学者推荐使用 `IDLE`。

Debug `Python` 代码

- 查看报错信息，根据信息修复代码。
- 插入 `print` 语句。记住发布代码之前要删除调试 `print` 语句或者把它们注释掉。
- 使用 `IDLE` 的图形用户界面调试器。其他 `IDE` 也有类似的调试器。
- 使用 `pdb` 命令行调试。`pdb` 是 `Python` 标准库中的一个模块。你可以交互的导入它，也可将它作为顶层文件运行。`pdb` 同样包括一个事后析误函数（`postmortem function`），`pdb.pm()`，你可以在异常发生之后运行，获取错误发生时的信息。第 36 章，`Python` 手册和附录 A 有关于 `pdb` 更详细的信息。
- 使用 `Python` 的 `-i` 命令行参数。`python -i m.py` 会让 `Python` 在你的程序退出时进入交互式解释器模式 `interactive interpreter`，不管程序有没有正常退出。这样你就可以打印一些信息，甚至可以导入 `pdb` 调试器，它的事后析误模式可以让你审查代码的最新错误（如果脚本运行失败的话）。附录 A 讲了 `-i` 的用法。
- 其他选择。比如 `Winpdb` 系统，是独立的调试器。

3.11 本章总结

本章学习了运行 `Python` 程序的常见方法：

- 交互式运行输入的代码
- 用命令行运行写好的文件
- 图标点击
- 模块导入
- `exec(open('module.py').read())`
- IDE GUI 比如 IDLE

3.12 检验你的知识：小测试

1. 怎样打开一个交互式解释器会话？
 - 开始菜单选择 `Python (command line)`
 - 系统命令行模式输入 `python`
 - 使用 `IDLE`
2. 在什么地方输入系统命令行来运行脚本文件？

在系统终端 `system console` 里面输入系统命令行，运行脚本文件。
3. 请说出运行保存在脚本文件里的代码的四种或者更多方式。
 - 系统命令行
 - 点击图标
 - `import` 和 `reload`
 - `exec` 内置函数
 - `IDE` 里面的运行选项
4. 说出在 `Windows` 平台上点击图标运行文件的两个陷阱。
 - 打印并退出的脚本会一闪而过（使用 `input` 来解决这个问题）
 - 报错信息会一闪而过（所以系统命令行、`IDE` 是更好的开发环境）
5. 为什么你需要 `reload` 一个模块？

`Python` 在一个进程中只导入 `import` 模块一次。

在不想停止并重启 `Python` 的情况下，想要运行改动后的源代码，就需要 `reload` 重新加载它。注意在 `reload` 之前必须先导入 `import` 一次。

使用其它方式运行文件则没有这个问题。
6. 在 `IDLE` 里面如何运行脚本？

`Run - Run File`
7. Name two pitfalls related to using IDLE.
8. 说出在用 `IDLE` 时的两个陷阱。
 - 运行一些程序的时候 `IDLE` 会被挂起。
 - 一些易用特性会在你不使用 `IDLE` 时造成困惑：比如运行的脚本的变量会被自动导入到交互模式；`IDLE` 在运行文件时会自动改变工作目录。
9. What is a namespace, and how does it relate to module files?
10. 什么是命名空间，它和模块文件有什么关系？

命名空间就是一些变量（即名字）的包 `package`。

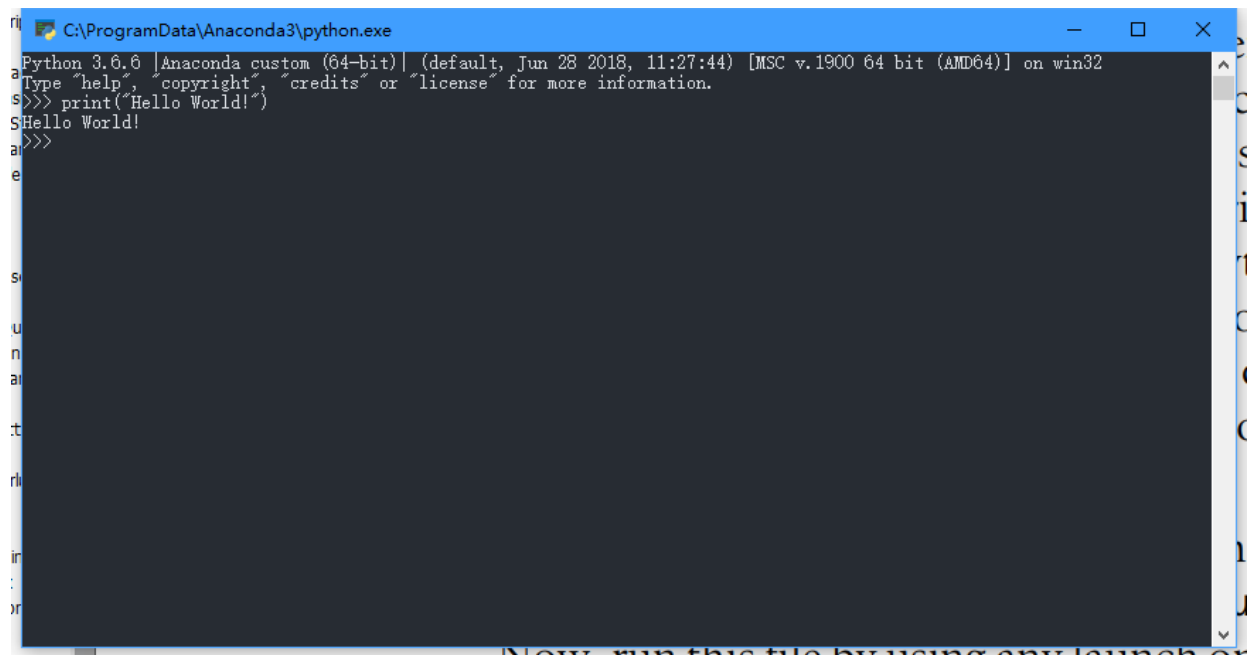
在 `Python` 中命名空间以拥有属性的对象的形式呈现。

每个模块文件自动形成一个命名空间。

命名空间可以避免命名冲突，因为每个模块都是一个独立的命名空间，使用别的文件的的名字必须显式的导入它们。

3.13 检验你的知识：第一部分练习

1. 交互。



```
C:\ProgramData\Anaconda3\python.exe
Python 3.6.6 |Anaconda custom (64-bit)| (default, Jun 28 2018, 11:27:44) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('Hello World!')
Hello World!
>>>
```

2. 程序。

编写 `module1.py`

```
print("Hello module world!")
```

用尽可能多的方式运行。

3. 模块。

用交互式命令行 `import` 在第 2 题中的程序。

移动该程序到其他目录，然后再 `import`，会出现什么情况？

假如 `module1.pyc` 存在的话，或者有类似 `__pycache__` 文件存在的话，是完全没有问题的。

4. 脚本。

```
#!/usr/local/bin/python
```

```
#!/usr/bin/env python
```

如果使用的是 `Unix` 系统，加入上面两行。

然后 `chmod +x module1.py`

最后 `module1.py` 回车运行。

[!] 在 `Windows` 的 `PowerShell` 中，可以用 `.\module1.py` 直接运行文件。

5. 错误与调试。

假如在交互命令行键入 `1 / 0` 或者引用没有定义的变量，会报错。

当你出错时，你就在进行错误处理，在这里触发了默认异常处理器 `default exception handler`。如果你不捕捉异常，异常就会被默认异常处理器捕捉，作为回应，它会打印标准错误信息。

6. 循环与中断。

在 `Python` 命令行模式中：

```
L = [1, 2] # Make a 2-item list
L.append(L) # Append L as a single item to itself
L # Print L: a cyclic/circular object
```

Ctrl-C 中断

【!】 Python 3.6 版本不需要中断，但是会有无穷多的嵌套列表。

7. 文档。

在一些 Windows 开始菜单能找到，在 IDLE 的 Help 下拉菜单里的 Python Docs 选项，或者在网上查阅 <http://www.python.org/doc>。第 15 章也会讲。

有时间看看 PyPI 第三方扩展库。特别关注 Python 官网 <http://www.python.org> 的文档和搜索页面，它们是很重要的资源。

完成于 201808062107