简明Python教程(中文版 2017)个人笔记

### 介绍

#### 1.1简明 Python 教程简介

1、**介绍如何使用Python语言进行编程**的自由图书。

2、它以**教材与指南**的形式**为入门者介绍 Python 语言**。

3、 **不需要编程知识**，如果你对电脑知识的了解仅限于**如何保存文本文件的话**，那这本书就是为你准备的。

 4、 本书将**指导你如何使用 Python 3**。

1. 本书也会以指南的形式告诉你应如何**适应**虽老些但**使用更加普遍的 Python 2**

7、 本书被各类教育机构当作他们的教材，如**哈佛大学** 的 Python 编程（Programming With Python） 课程

8、下载 你可以访问 https://www.gitbook.com/book/lenkimo/byte-of-python-chinese-edition/ 以获得本书以下格式的下载：

PDF （可在电脑上阅读）

EPUB （可在 iPhone、iPad、Mobi （可在 Kindle 上阅读）

#### 1.2本书的编写目的

1、**Python** 可能是极少数能够同时**兼顾简单与功能强大的编程语言**。无论是对于新手或是行家，这一点都裨益颇深。

2、**采用 Python 编程充满了乐趣**。

3、 本书旨在于**帮助你学习这一美妙的程序语言**，并向你展现**如何快速且毫不费力地完成诸多事情**，实际上也可说是“助你解决编程问题的抗毒血清”。

#### 1.3本书是为谁而撰

1、 本书以**新手**为**主要目标**。同时本书也对有**经验的**程序员**有所帮助**。

2、 如果你没有编程经验，那本书的目标便是协助你通过本书学习 Python。如果在此之前你已经有了编程经验，你同样可以**通过本书来学习 Python**。

3、 如果你已经**有过编程经验，**你或许会对 Python 与其它你所喜爱的编程语言间**有何区别抱有兴趣**——而我将会你展现许多这种区别。

#### 1.4官方网站

1、 本书**英文原版**的官方网站是 http://python.swaroopch.com/ ，在此你可以在线阅读本书的全部内容，下载本书的最新版本，购买本书的实体版，或是向我提交反馈。

2、 你现在阅读的这一**简体中文版**的刊载网站是 https://bop.molun.net ，你可以在此在线阅读本书的简体中文版，并下载最新版本。

#### 1.5 值得思考的一些事情

 1、 构建一项软件设计有两种方式：一种是**将软件设计得足够简单以至于明显找不到缺陷**；另一种是**软件设计得足够复杂以至于找不到明显的缺陷**。 ——查尔斯·安东尼·理查德·霍尔爵士（C. A. R. Hoare）

2、  **在人生中取得成功，与其说靠天才与机会，不如说靠专注与毅力**。 ——C. W. Wendte

### 关于 Python

#### 简介

1、 Python 是一种极少数能声言**兼具 简单 与 功能强大 的编程语言**。

2、 你将惊异于发现**你正在使用的这门编程语言是如此简单**，它**专注于如何解决问题，而非拘泥于语法与结构**。

3、 **官方对 Python 的介绍如下**：

         Python 是一款易于学习且功能强大的编程语言。 它具有高效率的数据结构，能够简单又有效地实现面向对象编程。Python 简洁的语法与动态输入之特性，加之其解释性语言的本质，使得它成为一种在多种领域与绝大多数平台都能进行脚本编写与应用快速开发工作的理想语言。

#### 名字背后的故事

Python 的创造者吉多·范罗苏姆（Guido van Rossum）采用 BBC 电视节目《蒙提·派森的飞行马戏团（Monty Python's Flying Circus，一译巨蟒剧团）》的名字来为这门编程语言命名。

尽管他本人并不特别喜欢蟒蛇这种通过在猎物身边卷曲自己的身体以此来碾碎猎物身体来进食的动物。

#### Python 的特色

##### 简单

1、 Python 是一门简单且简约的语言。

2、 阅读一份优秀的 Python 程序代码就如同在阅读英语文章一样，尽管这门英语要求十分严格！Python 这种伪代码式的特质正式它的一大优势。

3、 它能够**让你专注于解决问题的方案，而不是语言本身**。

##### 易于学习

Python 是一门非常容易入门的语言，**Python 有一套极其简单的语法体系**。

##### 自由且开放

1、 Python 是 **FLOSS （自由/开放源代码软件）的成员之一**。

2、 简单来说，你可以**自由地分发这一软件的拷贝，阅读它的源代码，并对其作出改动，或是将其的一部分运用于一款新的自由程序中**。

3、 FLOSS 基于一个**可以分享知识的社区理念而创建**。这正是 Python 为何能如此优秀的一大原因——它由一群希望看到 Python 能变得更好的社区成员所创造，并持续改进至今。

##### 高级语言

 当你在用 Python 编写程序时，你**不必考虑诸如你的程序应当如何使用内存等底层细节**。

##### 跨平台性

 你所有的 **Python 程序可以在其中任何一个平台上工作，不必作出任何改动**。

##### 解释性

1、 在你使用诸如**C 或 C++** 等编译语言编写程序时，需要将这些语言的源代码**通过编译程序配合中不同的标记（Flags）与选项，来将它们转换成你的电脑所使用的语言**（例如 0 与 1 构成的二进制码）。当你运行这些程序时，**链接程序或载入程序将会从硬盘中将程序拷贝至内存中并将其运行**。

2、 另一方面，**Python 不需要将其编译成二进制码**。你只需要直接**从源代码 运行 该程序**。

3、 在程序内部，**Python 会将源代码转换为称为字节码的中间形式，尔后再转换成你的电脑所使用的语言，并运行它**。

4、 实际上，**这一流程使得 Python 更加易于使用，你不必再担心该如何编译程序，或如何保证适当的库被正确的链接并加载等等步骤**。

5、 这也同样使得 Python 程序更**便携且易于迁移**，你只需要将 Python 程序拷贝到另一台电脑便可让它立即开始工作！

##### 面向对象

1、 Python 同时支持**面向过程编程**与**面向对象编程**。

2、 在 **面向过程**的编程语言中，程序是由仅仅带有可重用特性的子程序与函数所构建起来的。

3、 在 面向对象 的编程语言中，程序是由结合了数据与功能的对象所构建起来的。

4、 与 C++ 或 Java 这些大型语言相比，Python **具有其特别的、功能强大又简单**的方式来实现**面向对象编程**。

##### 可扩展性

 如果你需要代码的**某一重要部分能够快速地运行，或希望算法的某些部分不被公开**，你可以在**C 或 C++ 语言中编写**这些程序，然后再将其**运用于你的 Python 程序**中。

##### 可嵌入性

你可以**在你的 C 或 C++ 程序中嵌入 Python**，从而向你的程序用户提供 脚本 功能

##### 丰富的库

1、 实际上 **Python 标准库的规模非常庞大**。

2、 它能够帮助你完成诸多事情，包括正则表达式、文档生成、单元测试、多线程、数据库、网页浏览器、CGI、FTP、邮件、XML、XML-RPC、HTML、WAV 文件、密码系统、GUI（图形用户界面），以及其它系统依赖型的活动。

3、 只需记住，**只要安装了 Python，这些功能便随时可用**。它们的存在被称作 Python 自备电池（*Batteries Included*） 式的哲学。

4、 除了标准库以外，你还可以在 **Python 库索引**（Python Package Index） 中发掘许多其它高质量的库。

##### 总结

Python 着实是**一门令人心生激动且强大的语言**。它得当地结合了性能与功能，使得编写Python 程序是如此简易又充满乐趣。

#### Python 3 VS Python 2

1、**本书是以 Python 3 为对象撰写的。**

2、 只消记住**一旦你正确理解并学习了其中一个版本的 Python**，**你便可以很容易地理解另一版本的区别，并能快速学习如何使用**。

3、**困难的是学习如何编程以及理解 Python 语言本身的基础部分**。这便是我们在本书中的目标，而一旦你达成了目标，你便可以根据你的实际情况，决定是该使用 Pyhon 2 还是 Python 3。

### 安装

我们在本书中提及“Python 3”时，我们指的是**任何大于等于该版本的 Python 3.5.1**。

#### 在 Windows 中安装

1、 访问 https://www.python.org/downloads/ 并下载最新版本的 Python。在本书撰写的时点，最新版本为 Python 3.5.1。

2、 其安装过程与其它 Windows 平台的软件的安装过程无异。

3、 注意：**请务必确认你勾选了 Add Python 3.5 to PATH 选项**。

4、 若要想改变**安装位置**，勾选 Customize installation 选项，点击 Next 后在安装位置中输入C:\python35 。

5、 如**未勾选相关选项**，你可以点击 Add Python to environment variables 。它和安装程序第一屏的 Add Python 3.5 to PATH 能起到相同效果。

6、 你可以选择是否为所有用户安装启动器，这不会产生多大影响。启动器用以切换已安装的不同版本的 Python。

7、 如果你的环境变量（Path）未正确设置，可以遵循上述步骤予以修正。否则，请参阅 在Windows 中运行 Python 提示符 。

注意：对于那些对编程有所了解的人，如果你熟悉 Docker，可以参阅 Python in Docker 和Docker on Windows。

#### DOS 提示符

1、 如果你希望在 Windows 命令行使用 Python，比如 DOS 提示符，你需要**设置相应的PATH 境变量**。

2、 对于 Windows 2000，XP，2003，点击 控制面板 → 系统 → 高级 → 环境变量 。点击 系统变量 选项卡下名为 PATH 的变量，选择 编辑 并在已存在的字符串末尾添加 ;C:\Python35（请确保该文件夹确实存在，对于更新版本的 Python 文件夹的名字可能有所不同）。当然，你应该使用恰当的目录名称。

3、 对于更古老的 Windows 系统版本，打开文件 C:\AUTOEXEC.BAT 并在其中添加一行PATH=%PATH%;C:\Python35 ，完成编辑并保存后，你需要重启系统。对于**Windows NT 系统**，则对应 AUTOEXEC.NT 文件。

4、 对于 **Windows Vista**：

点击开始并选择 控制面板 。

点击系统，在右侧你将会看见“浏览你的计算机的基本信息”

左侧将由一个任务列表，最后一个即为“高级系统设置”，点击它。

高级 标签栏下可以看见 系统属性 对话框。

在较低位置的名为 系统变量 的对话框，向下滚动至变量部分并点击 编辑 按钮。

修改你需要改动的变量。

重启系统。Vista 直至重启前都不会应用系统变量环境的改动。

5、 对于**Windos 7 与 8**：

在桌面右击计算机并选择 属性 或点击 开始 并选择 控制面板 → 系统与安全 → 系统 。

点击左侧的 高级系统设置 并选择 高级 标签。点击底部 系统变量 下的 环境变量 ，找到

PATH 属性，将其选中并点击 编辑 。

前往变量值的最后一行并添加 ;C:\Python35 （请确保该文件夹确实存在，对于更新版本的 Python 文件夹的名字可能有所不同）至业已存在的部分的后方。当然，你应该使用恰当的目录名称。

如果该变量值为 %SystemRoot%\system32; 则在修改后其应变为%SystemRoot%\system32;C:\Python35 。

点击 确定 以完成操作。你不需要进行重启，不过你可能需要关闭并重启命令提示符。

#### 在 Windows 下运行 Python 命令提示符

1、 对于 Windows 用户来说，如果你已经正确并恰当**地设置了 PATH 变量**，你可以在命令行中运行解释程序。

2、 要想在 Windows 中运行终端，点击开始并点击 运行 。在对话中输入 cmd 并按下回车键。然后，输入 python 以确保其没有任何错误。

#### 在 Mac OS 下安装

#### 在 GNU/Linux 下安装

1、 对于 GNU/Linux 用户，你可以**使用发行版的包管理器来安装 Python 3**，例如在 Debian 与Ubuntu 平台下，你可以输入命令： sudo apt-get update && sudo apt-get install python3 。

2、 要想验证**安装是否成功**，你可以通过打开 Terminal 应用或通过按下 Alt + F2 组合键并输入 gnome-terminal 来启动终端程序。

3、 如果这不起作用，请查阅你**所使用的的 GNU/Linux 发行版**的文档。现在，运行 python3 命令来确保其没有任何错误。

4、 你会看到在运行命令后 Python 的版本信息显示在屏幕上：

           $ **python3 -V**

**Python 3.5.1**

### 第一步

1、如何**编写、保存与运行 Python 程序**，示例如何在 Python 中运行一个传统的“Hello World”程序。

2、 通过 Python 来运行的你的程序有**两种方法**—

         1、 使用**交互式解释器**提示符

         2、 直接**运行一个源代码**文件。

#### 使用解释器提示符

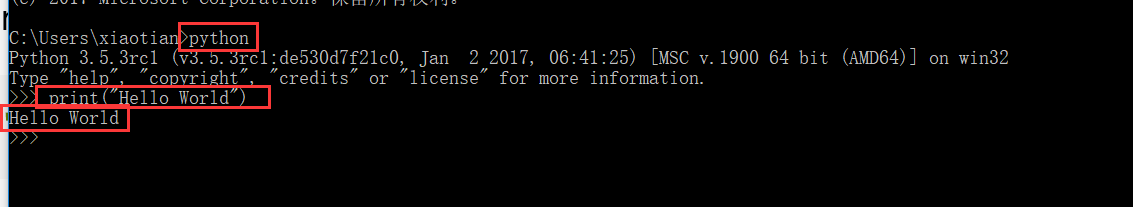
1、 在你的操作系统中**打开终端**（Terminal）程序

2、 然后通过输入 python3 并按下 [enter] 键来**打开 Python 提示符**（Python Prompt）。

3、 当你**启动 Python 后**，你会看见在你能开始输入内容的地方**出现了**>>> 。

4、  在 Python 解释器提示符，输入： print("Hello World")

5、  在输入完成后按下 [enter] 键。你将会看到屏幕上打印出 Hello World 字样。



#### 如何退出解释器提示符

1、 如果你正在使用一款 GNU/Linux 或 OS X 上的 Shell 程序，你可以通过**按下** [ctrl + d] 组合键或是**输入** exit() （注意：要记住要包含括号 () ）并敲下 [enter] 来退出解释器提示符。

2、 如果你使用的是 Windows 命令提示符，可以按下 [ctrl + z] 组合键并敲击 [enter] 键来退出。

#### 选择一款编辑器

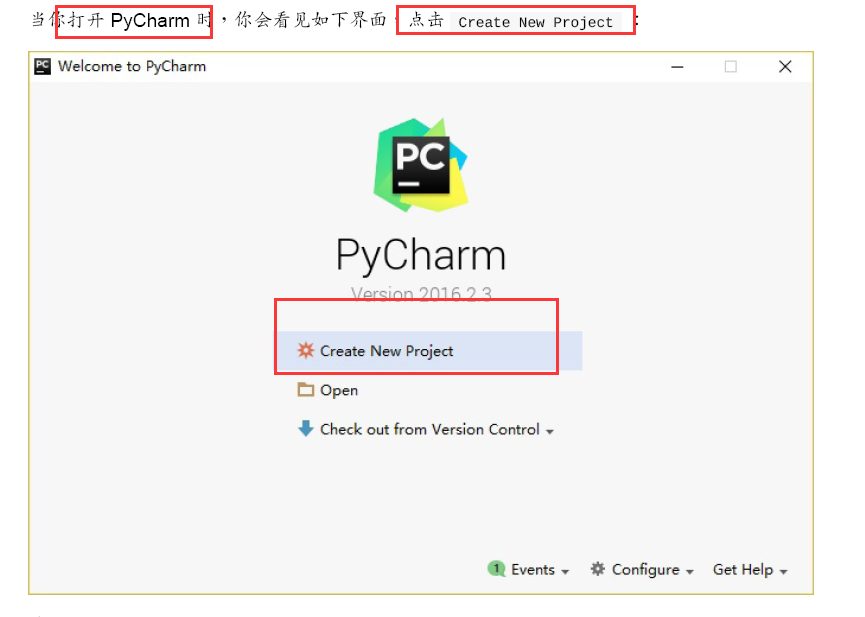
1、 一款优秀的面向程序员的编辑器能够帮助你的**编写源代码文件工作变得轻松得多**。

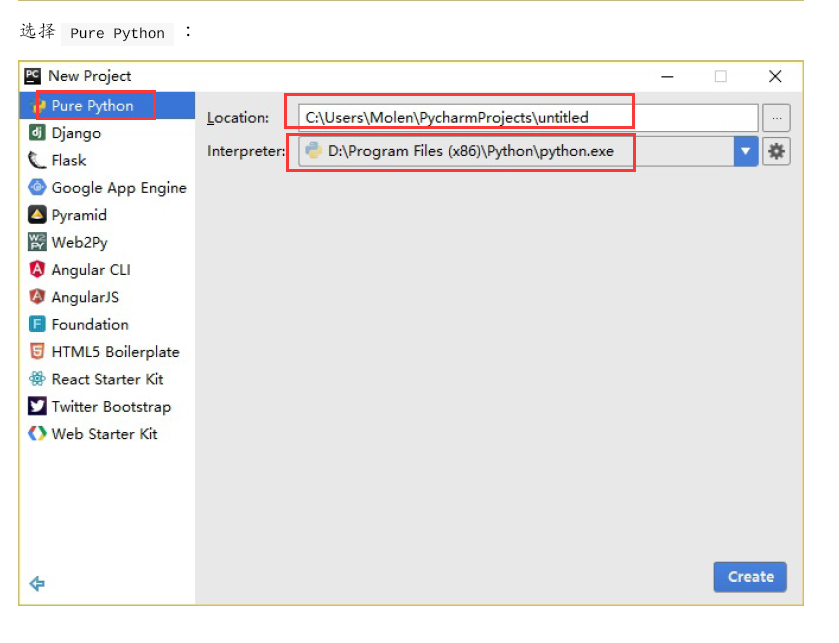
2、 一款优秀的编辑器能够帮助你**更轻松地编写 Python 程序，使你的编程之旅更加舒适**，并助你找到一条更加安全且**快速的道路到达你的目的地**（实现你的目标）。

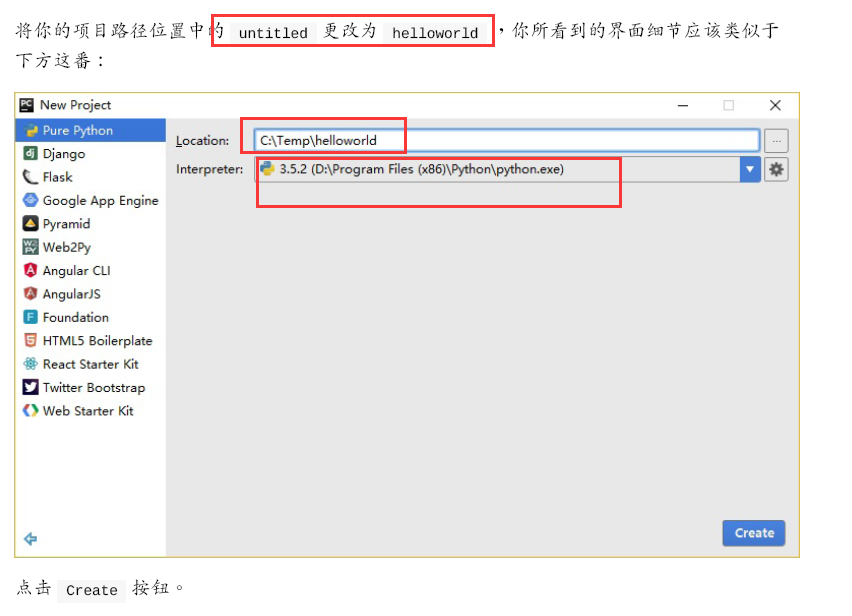
3、 对编辑器的一项最基本要求为**语法高亮** ，这一功能能够通过标以不同颜色来帮助你区分Python 程序中的不同部分，**从而能够让你更好 看清 你的程序，并使它的运行模式更加形象化**。

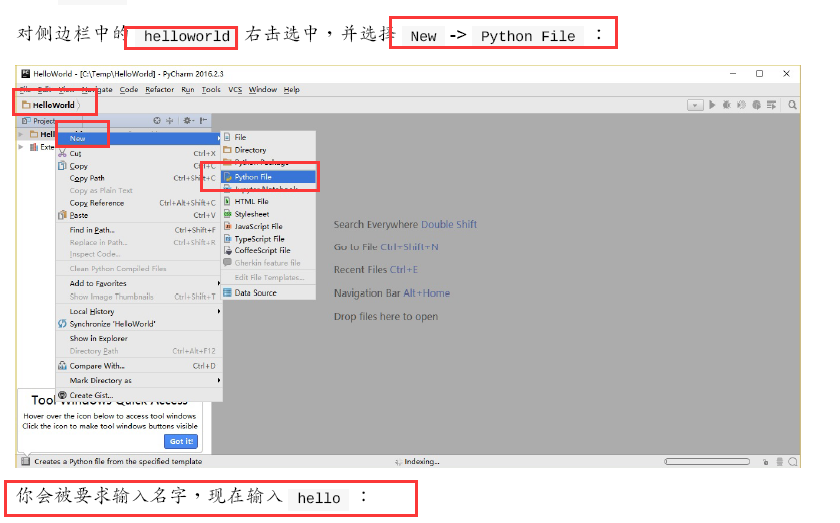
##### PyCharm

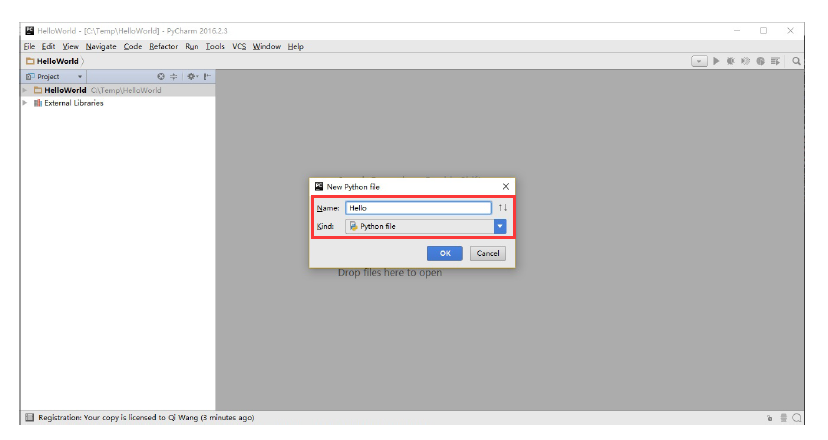
1、 PyCharm 教育版是一款能够对你编写 Python 程序的工作有所帮助的**免费编辑器**。

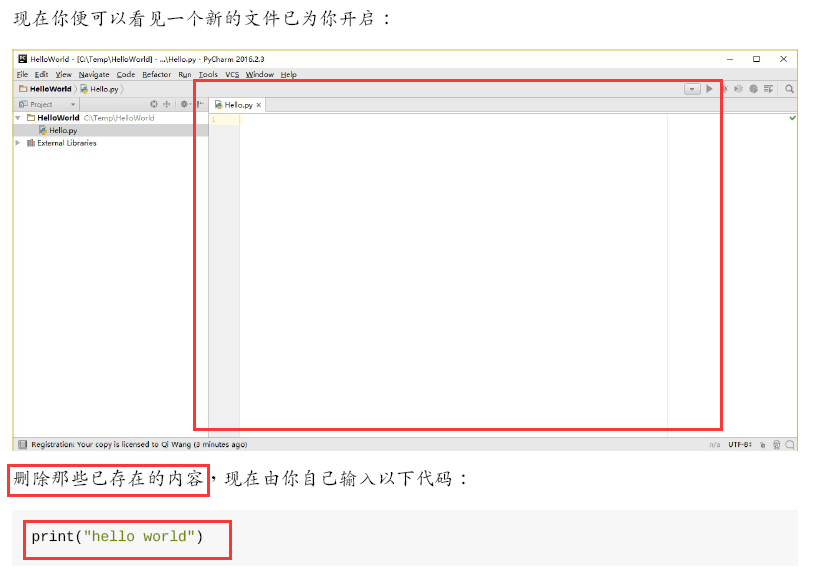


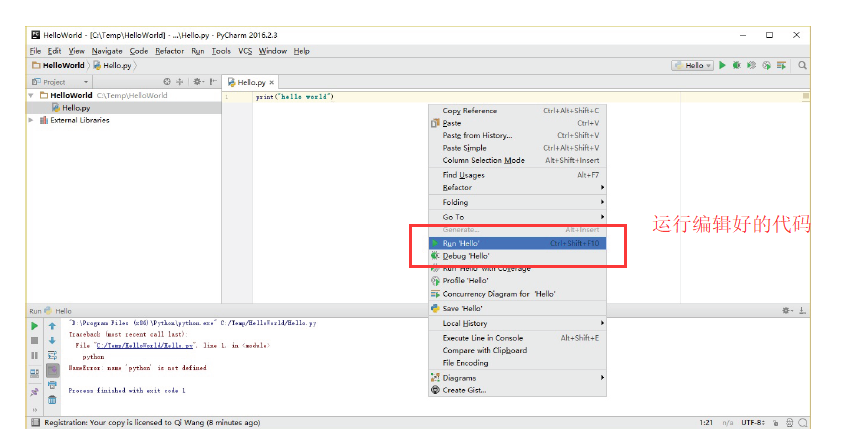


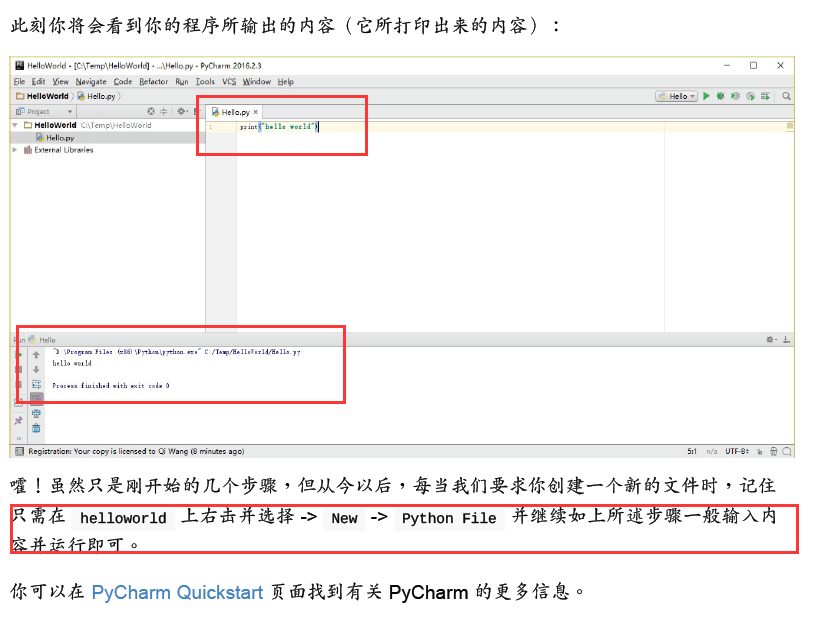












##### Vim

1. **安装** Vim。

Mac OS X 应该通过 HomeBrew 来安装 macvim 包。

Windows 用户应该通过 Vim 官方网站 下载“自安装可执行文件”。

GNU/Linux 用户应该通过他们使用的发行版的软件仓库获取 Vim。例如 Debian 与

Ubuntu 用户可以安装 vim 包。

2. **安装 jedi-vim 插件**为 Vim 增添自动完成功能。

3. **安装与之相应的 jedi Python 包**： pip install -U jedi

##### Emacs

1. **安装**Emacs 24+。

Mac OS X 用户应该从 http://emacsformacosx.com 获取 Emacs。

Windows 用户应该从 http://ftp.gnu.org/gnu/emacs/windows/ 获取 Emacs。

GNU/Linux 用户应该从他们使用的发行版的软件仓库获取 Emacs。如 Debian 和

Ubuntu 用户可以安装 emacs24 包。

2. **安装 ELPY。**

#### 使用一份源代码文件

1、 **启动你所选择的编辑器**，输入如下程序并将它保存为 hello.py 。

         1）如果你正在使用 PyCharm，我们已经讨论过如何从源文件中运行它了。

        2）对于其它编辑器，打开一个新文件名将其命名为 hello.py ，然后输入如下内容：

**print("hello world")**

2、 **你应当将文件保存到哪里**？保存到任何你知道其位置与路径的文件夹。如果你不了解这句话是什么意思，那就创建一个新文件夹并用这一路径来保存并运行你所有的 Python 程序：

**Mac OS X** 上的 /tmp/py 。

**GNU/Linux** 上的 /tmp/py 。

**Windows** 上的 C:\\py 。

3、 要想**创建上述文件夹**（在你正在使用的操作系统上），你可以在终端上**使用 mkdir 命令**，如mkdir /tmp/py 。

4、 **重要提示**：你需要经常确认并确保你为文件赋予了 .py 扩展名，例如 foo.py 。

5、 要想运行你的 Python 程序：

    1. **打开终端窗口**（你可查阅先前的 安装章节来了解应该怎么做）。

    2. **使用 cd 命令来改变目录到你保存文件的地方**，例如 cd /tmp/py 。

    3. 通过**输入命令 python hello.py 来运行程序**。程序的输出结果应如下方所示：

                $ python hello.py

6、 如果你**遭遇了什么错误**，请确认是否已经正确地上面所列出的内容并重新运行程序。要注意Python 是区分大小写的，如 print 和 Print 是不同的——注意前者的 p 是小写的，而后者的 P 是大写的。此外，你需要确保**每一行的第一个字符前面都没有任何空格或制表格**。

7、 它是如何工作的

         一款 Python 程序是**由 语句 所构成**的。在我们的第一个程序中，我们只有一条语句。在这条语句中，我们调用 print 语句来搭配我们提供的文本”hello world“。

#### 获取帮助

1、 如果你需要获得 Python 中有**关任何函数或语句的快速信息**，你可以使用其内置的**help 功能**。这在使用解释器提示符时十分有用。

2、 例如，**运行 help('len') 命令**——这将显示出有**关len 函数的帮助**，了解其是用来计算项目数量的。

3、 小贴士：**按下 q 键可以退出帮助**。

4、 类似地，你可以通过此方式**获得几乎所有有关 Python 的信息**。**使用 help() 命令**来了解有关help 它本身的更多信息吧！

5、 如果你需要获得**有关 return 这类运算符的帮助**，你**需要做的就是将它们放在引号中**，就像help('return') 这般，这样 Python 就不会混淆我们正在试图做的事情。

1. 基础

1、 只是打印出 hello world 肯定是不够的，是吗？你会希望做得比这还要多——你想要**输入一些内容，操纵它，然后从中得到一些输出出来的内容**。

2、 我们可以在 Python 中通过使用**变量与常量**来实现这一目标，在本章中我们还会学习其它的一些概念。

* 1. 注释

1、 **注释** 是任何存在于 # 号右侧的文字，其主要用作**写给程序读者看的笔记**。

2、 举个例子：

           print('hello world') #注意到 print 是一个函数

         或者：

**# 注意到 print 是一个函数**

           print('hello world')

3、 你应该**在你的程序中尽可能多地使用有用的注释**：

         1）解释假设

         2）说明重要的决定

         3）解释重要的细节

         4）说明你想要解决的问题

         5）说明你想要在程序中克服的问题，等等。

4、 **代码会告诉你怎么做，注释会告诉你为何如此**。

5、 这样做**对你的程序的读者来说非常有用**，他们可以**很容易地理解你的程序是做什么的**。请记住，**这个人可以是六个月后的你！**

* 1. 字面常量

1、 一个**字面常量**（Literal Constants） 的例子是诸如 5 、 1.23 这样的**数字**，或者是如 这是一串文本 或 This is a string 这样的**文本**。

2、 这么称呼这一变量是因为它们是 **字面上的** ——**你用的就是它字面意义上的值或是内容**。数字 2 总是表示它本身而非其他含义——**它是一个 常量，因为它的值不能被改变**。因此，所有的这些都被称作字面常量。

* 1. 数字

1、 数字主要分为**两种类型**——**整数**（Integers）与**浮点数**（Floats）。

2、 有关**整数**的例子即 2 ，它只是一个整数。

3、 有关**浮点数**（Floating Point Numbers，在英文中也会简写为 *floats* ）的例子是 3.23 或52.3E-4 。其中， E 表示 **10 的幂**。在这里， 52.3E-4 表示 52.3 \* 10^-4 。

* 1. 字符串

1、 **一串字符串**（String）是 字符（*Characters*） 的 序列（*Sequence*）。

2、 基本上，**字符串就是一串词汇**。

3、 你**将会在几乎所有你撰写的 Python 程序中使用字符串**，所以对下面的部分你要多上点心。

* + 1. 单引号

1、 你可以使用**单引号**来指定字符串，例如 '将我这样框进来' 或 'Quote me on this' 。

2、 **所有引号内的空间**，诸如空格与制表符，**都将按原样保留**。

* + 1. 双引号

1、 被**双引号**包括的字符串和被**单引号**括起的字符串其工作机制**完全相同**。

2、 例如 "你的名字是？" 或 "What's your name?"

* + 1. 三引号

1、 你可以通过**使用三个引号**——**""" 或 ''' 来指定多行字符串**。

2、 你可以在**三引号之间自由地使用单引号与双引号**。

3、 来看看这个例子：

'''这是一段多行字符串。这是它的第一行。

This is the second line.

"What's your name?," I asked.

He said "Bond, James Bond."

'''

* 1. 字符串是不可变的

1、这意味着**一旦你创造了一串字符串，你就不能再改变它**。

2、 尽管这**看起来像是一件坏事，但实际上并非如此**。

3、 我们将会在稍后展现的多个程序中看到**为何这一点不是一个限制**。

* 1. 格式化方法

1、 有时候我们会想要**从其他信息中构建字符串**。这**正是 format() 方法**大有**用武之地**的地方。

2、 将以下内容保存为文件 str\_format.py ：

           age = 20

           name = 'Swaroop'

           print('{0} was {1} years old when he wrote this book'.format(name, age))

           print('Why is {0} playing with that python?'.format(name))

         输出：

           $ python str\_format.py

           Swaroop was 20 years old when he wrote this book

           Why is Swaroop playing with that python?

3、 它是**如何工作**的

**一个字符串可以使用某些特定的格式（Specification），随后， format 方法将被调用，使用这一方法中与之相应的参数替换这些格式。**

         在这里要注意我们**第一次应用这一方法的地方**，此处 {0} 对应的是变量 name ，它是该格式化方法中的第一个参数。与之类似，第二个格式 {1} 对应的是变量 age ，它是格式化方法中的第二个参数。请注意，**Python 从 0 开始计数，这意味着索引中的第一位是 0，第二位是1，以此类推**。

4、 我们可以**通过联立字符串**来达到相同的效果：name + 'is' +str(age) + 'years old'

         1）但这样**实现是很丑陋的，而且也容易出错**。

         2）其次，**转换至字符串的工作将由 format 方法自动完成**，而不是如这般需要明确转换至字符串。

         3）再次，当时**用 format 方法**时，我们可以直接改动文字而不必与变量打交道，反之亦然。

5、 同时还应注意**数字只是一个可选选项**，所以你同样可以写成：

           age = 20

           name = 'Swaroop'

           print('{} was {} years old when he wrote this book'.format(name, age))

           print('Why is {} playing with that python?'.format(name))

         这样做同样能得到与前面的程序一样的输出结果。

6、 Python 中**format 方法**所做的事情便是**将每个参数值替换至格式所在的位置**。这之中可以有更详细的格式，例如：

           # 对于浮点数 '0.333' 保留小数点(.)后三位

           print('{0:.3f}'.format(1.0/3))

           # 使用下划线填充文本，并保持文字处于中间位置

           # 使用 (^) 定义 '\_\_\_hello\_\_\_'字符串长度为 11

           print('{0:\_^11}'.format('hello'))

           # 基于关键词输出 'Swaroop wrote A Byte of Python'

           print('{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python'))

         输出：

           0.333

           \_\_\_hello\_\_\_

           Swaroop wrote A Byte of Python

7、 由于我们正在讨论格式问题，就要注意 print 总是会**以一个不可见的“新一行”字符（ \n ）结尾**，因此**重复调用 print 将会在相互独立的一行中分别打印**。为防止打印过程中出现这一换行符，你可以通过 end 指定其应以空白结尾：

           print('a', end='')

           print('b', end='')

         输出结果如下：

                 ab

         或者你通过 end 指定以空格结尾：

           print('a', end=' ')

           print('b', end=' ')

           print('c')

         输出结果如下：

                 a b c

* 1. 转义序列

1、 想象一下，如果你希望**生成一串包含单引号（ ' ）的字符串**，你应该如何指定这串字符串？

2、 例如，你想要的字符串是 "What's your name?" 。你**不能**指定 'What's your name?' ，因为这会**使 Python 对于何处是字符串的开始、何处又是结束而感到困惑**。

3、 所以，你必须指定这个**单引号不代表这串字符串的结尾**。这可以通过**转义序列**（*Escape Sequence*） 来实现。你通过\ 来指定单引号：要注意它可是**反斜杠**。现在，你可以将字符串指定为 'What\'s yourname?' 。

4、 另一种指定这一特别的字符串的方式是这样的： "What's your name?" ，如这个例子般使用双引号。类似地， 你必须在使用双引号括起的字符串中对字符串内的双引号使用转义序列。

5、 同样，你必须使用转义序列**\\ 来指定反斜杠本身**。

6、 如果你想**指定一串双行字符串**该怎么办？一种方式即**使用**如前所述的**三引号字符串**，或者你可以**使用一个表示新一行的转义序列—— \n 来表示新一行的开始**。下面是一个例子：

           'This is the first line\nThis is the second line'

7、 另一个你应该知道的大有用处的转义序列是**制表符**：**\t** 。实际上还有很多的转义序列，但

我必须只在此展示最重要的一些。

8、 还有一件需要的事情，**在一个字符串中，一个放置在末尾的反斜杠表示字符串将在下一行继**

**续，但不会添加新的一行。**来看看例子：

           "This is the first sentence. \

           This is the second sentence."

         相当于

       "This is the first sentence. This is the second sentence."

* 1. 原始字符串

1、 如果你需要**指定一些未经过特殊处理的字符串**，比如转义序列，那么你需要在字符串**前增加r 或 R 来指定一个 原始（*Raw*） 字符串**。

2、 下面是一个例子：

           r"Newlines are indicated by \n"

3、 针对**正则表达式用户**的提示

         在处理正则表达式时应全程使用原始字符串。否则，将会有大量 Backwhacking 需要处理。举例说明的话，反向引用可以通过 '\\1' 或 r'\1' 来实现。

* 1. 变量

1、 如果只使用字面常量很快就会让人感到无聊——我们**需要一些能够存储任何信息并且也能操纵它们的方式**。这便是**变量**（*Veriables*） 登场的时刻。

2、 正如其名字所述那般，**变量的值是可以变化的**，也就是说，你可以用变量来存储任何东西。

3、 变量只是你的**计算机内存中用以存储信息的一部分**。

4、 与文字常量不同，**你需要通过一些方式来访问这些变量**，因此，**你需要为它们命名**。

* + 1. 标识符命名

1、 变量是标识符的一个例子。标识符（*Identifiers*） 是**为 某些东西 提供的给定名称**。

2、 在你命名标识符时，你**需要遵守以下规则**：

         1）第一个字符必须是字母表中的字母（大写 ASCII 字符或小写 ASCII 字符或 Unicode 字符）或下划线（ \_ ）。

         2）标识符的其它部分可以由字符（大写 ASCII 字符或小写 ASCII 字符或 Unicode 字符）、下划线（ \_ ）、数字（0~9）组成。

         3）标识符名称区分大小写。例如， myname 和 myName 并不等同。要注意到前者是小写字母 n 而后者是大写字母 N 。

         4） 有效 的标识符名称可以是 i 或 name\_2\_3 ，**无效 的标识符**名称可能是2things ， this is spaced out ， my-name 和 >a1b2\_c3 。

* + 1. 数据类型

1、 **变量**可以将各种形式的值保存为不同的数据类型（*Data Type*）。

2、 **基本的类型**是我们已经讨论过的数字与字符串。

3、 在后面的章节中，我们会了解如何通过**类（Classes）** **类创建我们自己的类型**

* 1. 对象

1、 Python 将程序中的**任何内容统称为 对象（*Object*）**。

2、 这是一般意义上的说法。我们以“某某对象（*object*）”**相称**，而非“某某东西（*something*）”。

3、 针对面向对象编程语言用户的提示：

         Python 是强（Strongly）面向对象的，因为所有的一切都是对象， 包括数字、字符串与函数。

* 1. 如何编写 Python 程序
     1. 对于 PyCharm 用户

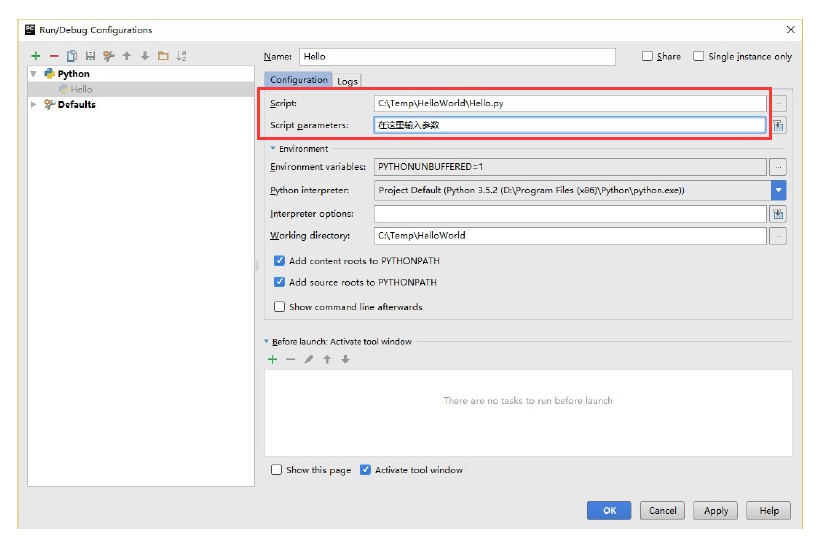
1. **打开 PyCharm。**

2. 以给定的文件名**创建新文件**。

3. **输入**案例中给出的**代码**。

4. **右键并运行当前文件**。

5、**注意**：每当你需要提供**命令行参数（Command Line Arguments）时**，点击 Run -> EditConfigurations 并在 Script parameters: 部分输入相应参数，并点击 OK 按钮：



* + 1. 对于其他编辑器用户

1. 打开你选择的编辑器。

2. **输入案例中给出的代码**。

3. 以给定的文件名将其保存成文件。

4. 在**解释器中通过命令 python program.py 来运行程序**。

* + 1. 案例：使用变量与字面常量

1、 输入并运行以下程序：

           # 文件名：var.py

           i = 5

           print(i)

           i = i + 1

           print(i)

           s = '''This is a multi-line string.

           This is the second line.'''

           print(s)

2、 输出：

           5

           6

           This is a multi-line string.

           This is the second line.

3、 它是**如何工作**的

         下面是这一程序的工作原理。首先，我们使用赋值运算符（ = ）将字面常量数值 5 赋值给

变量 i 。这一行被称之为**声明语句（Statement）因为其工作正是声明一些在这一情况下应**

**当完成的事情**：我们将变量名 i 与值 5 相连接。

         然后，我们通过 print 语句来打印变量i 所声明的内容，这并不奇怪，只是将变量的值打印到屏幕上。

接着，我们将 1 加到 i 变量所存储的值中，并将得出的结果重**新存储进这一变量**。尔后我们将这一变量**打印出来**，并期望得到的值应为 6 。

4、 类似地，我们将字面文本赋值给变量 s ，并将其打印出来。

针对静态编程语言程序员的提示

变量只需被赋予某一值。不需要声明或定义数据类型。

* 1. 逻辑行与物理行

1、 所谓**物理行（Physical Line）**是你**在编写程序时 你所看到 的内容**。

2、 所谓**逻辑行（LogicalLine）**是 ***Python* 所看到 的单个语句**。

3、 Python 会**假定每一 物理行 会对应一个 逻辑行**。

4、 有**关逻辑行的一个例子是诸如 print('hello world') 这样一句语句**——**如果其本身是一行（**正如你在编辑器里所看到的那样），那么它也对应着一行物理行。

5、 Python 之中暗含这样一种期望：**Python 鼓励每一行使用一句独立语句从而使得代码更加可读**。

6、 如果你希望**在一行物理行中指定多行逻辑行**，那么你必须通过**使用分号( ; )**来明确表明逻辑行或语句的结束。下面是一个例子：

           i = 5

           print(i)

         实际上等同于

           i = 5;

           print(i);

         同样可以看作

           i = 5; print(i);

         也与这一写法相同

           i = 5; print(i)

7、 然而，**我强烈建议你对于每一行物理行最多只写入一行逻辑行**。这个观点就是说你**不应该使用分号**。实际上，我从未在 Python 程序中使用、甚至是见过一个分号。

8、 在一类情况下这一方法会颇为有用：**如果你有一行非常长的代码，你可以通过使用反斜杠将**

**其拆分成多个物理行。这被称作显式行连接（*Explicit Line Joining*）** ：

           s = 'This is a string. \

           This continues the string.'

           print(s)

         输出：

                This is a string. This continues the string.

         类似地，

           i = \

           5

         等同于

           i = 5

9、 在某些情况下，会存在一个隐含的假设，**允许你不使用反斜杠。**这一情况**即逻辑行以括号开**

**始，它可以是方括号或花括号，但不能是结束括号。**这被称作**隐式行连接**（*Implicit Line*

*Joining*）。你可以在后面当我们讨论列表（List）的章节时了解这一点。

* 1. 缩进

1、 **空白区**在 Python 中十分重要。实际上，**空白区在各行的开头非常重要**。这被称作 **缩进**

（*Indentation*）。

2、在**逻辑行的开头**留下空白区（使用空格或制表符）用以确定各逻辑行的缩进级别，而后者又可用于确定语句的分组。

3、 这意味着**放置在一起的语句必须拥有相同的缩进**。每一组这样的语句被称为 块（*block*）。我们将会在后文章节的案例中了解块这一概念是多么重要。

4、 有一件事你需要记住：**错误的缩进可能会导致错误**。下面是一个例子：

           i = 5

           # 下面将发生错误，注意行首有一个空格

            print('Value is', i)

           print('I repeat, the value is', i)

         当你运行这一程序时，你将得到如下错误：

           File "whitespace.py", line 3

           print('Value is', i)

           ^

           IndentationError: unexpected indent

           # 缩进错误：意外缩进

5、 你会注意到第二行开头有一个空格。Python 指出的错误信息告诉我们**程序的语法是无效的**，意即，程序没有被正确地写入。这一信息对你的意义是 **你不能任意开始一个新的语句块**（当然，除非你一直在使用默认的主代码块）。你可以使用新块的情况将会在后面诸如控制流等章节加以介绍。

6、 **如何缩进**

**使用四个空格来缩进。这是来自 Python 语言官方的建议**。

    好的编辑器会自动为你完成这一工作。请确保你在缩进中使用数量一致的空格，否则你的程序将不会运行，或引发不期望的行为。

针对静态编程语言程序员的提示

Python 将始终对块使用缩进，并且绝不会使用大括号。你可以通过运行

from\_\_future\_\_import braces 来了解更多信息。

* 1. 总结

现在我们已经了解了诸多本质性的细节，我们可以前去了解控制流语句等更多更加有趣的东西。记得一定要充分理解你在本章所阅读的内容。

1. 运算符与表达式

1、你所编写的大多数语句（逻辑行）都包含了**表达式**（*Expressions*）。一个表达式的简单例子便是 2+3 。

2、 **表达式**可以拆分成**运算符**（Operators）与**操作数**（Operands）。

3、 **运算符**（*Operators*）是进行某些操作，并且可以用诸如 + 等符号或特殊关键词加以表达的功能。

4、 运算符需要一些**数据**来进行操作，这些数据就被称作操作数（*Operands*）。在上面的例子中 2 和 3 就是操作数。

* 1. 运算符

1、 要记得你可以**随时在解释器中对给出的案例里的表达式进行求值**。

2、 例如要想测试表达式 2+3 ，则可以**使用交互式 Python 解释器**提示符：

           >>> 2 + 3

           5

           >>> 3 \* 5

           15

           >>>

3、 下面是可用运算符的速览：

+ （加）

两个对象相加。

3+5 则输出 8 。 'a' + 'b' 则输出 'ab' 。

- （减）

从一个数中减去另一个数，如果第一个操作数不存在，则假定为零。

-5.2 将输出一个负数， 50 - 24 输出 26 。

\* （乘）

给出两个数的乘积，或返回字符串重复指定次数后的结果。

2 \* 3 输出 6 。 'la' \* 3 输出 'lalala' 。

**\*\* （乘方）**

返回 x 的 y 次方。

3 \*\* 4 输出 81 （即 3 \* 3 \* 3 \* 3 ）。

/ （除）

x 除以 y

13 / 3 输出 4.333333333333333 。

**// （整除）**

x 除以 y 并对结果向下取整至最接近的整数。

13 // 3 输出 4 。

-13 // 3 输出 -5 。

% （取模）

返回除法运算后的余数。

13 % 3 输出 1 。 -25.5 % 2.25 输出 1.5 。

<< （左移）

将数字的位向左移动指定的位数。（每个数字在内存中以二进制数表示，即 0 和1）

2 << 2 输出 8 。 2 用二进制数表示为 10 。

向左移 2 位会得到 1000 这一结果，表示十进制中的 8 。

>> （右移）

将数字的位向右移动指定的位数。

11 >> 1 输出 5 。

11 在二进制中表示为 1011 ，右移一位后输出 101 这一结果，表示十进制中的5 。

& （按位与）

对数字进行按位与操作。

5 & 3 输出 1 。

| （按位或）

对数字进行按位或操作。

5 | 3 输出 7 。

^ （按位异或）

对数字进行按位异或操作。

5 ^ 3 输出 6 。

**~ （按位取反）**

**x 的按位取反结果为 -(x+1)。**

~5 输出 -6 。有关本例的更多细节可以参阅：http://stackoverflow.com/a/11810203。

< （小于）

返回 x 是否小于 y。所有的比较运算符返回的结果均为 True 或 False 。请注意这些名称之中的大写字母。

5 < 3 输出 False ， 3 < 6 输出 True 。

比较可以任意组成组成链接： 3 < 5 < 7 返回 True 。

> （大于）

返回 x 是否大于 y。

5 > 3 返回 True 。如果两个操作数均为数字，它们首先将会被转换至一种共同的

类型。否则，它将总是返回 False 。

<= （小于等于）

返回 x 是否小于或等于 y。

x = 3; y = 6; x<=y 返回 True 。

>= （大于等于）

返回 x 是否大于或等于 y。

x = 4; y = 3; x>=3 返回 True 。

== （等于）

比较两个对象是否相等。

x = 2; y = 2; x == y 返回 True 。

x = 'str'; y = 'stR'; x == y 返回 False 。

x = 'str'; y = 'str'; x == y 返回 True 。

!= （不等于）

比较两个对象是否不相等。

x = 2; y = 3; x != y 返回 True 。

not （布尔“非”）

如果 x 是 Ture ，则返回 False 。如果 x 是 False ，则返回 True 。

x = Ture; not x 返回 False 。

**and （布尔“与”）**

如果 x 是 False ，则 x and y 返回 False ，否则返回 y 的计算值。

当 x 是 False 时， x = False; y = True; x and y 将返回 False 。在这一情境中，Python 将不会计算 y，因为它已经了解 and 表达式的左侧是 False ，这意味着整个表达式都将是 False 而不会是别的值。**这种情况被称作短路计算（Short-circuit Evaluation）。**

or （布尔“或”）

如果 x 是 True ，则返回 True ，否则它将返回 y 的计算值。

x = Ture; y = False; x or y 将返回 Ture 。**在这里短路计算同样适用。**

* 1. 数值运算与赋值的快捷方式

1、 一种比较常见的操作是对**一个变量进行一项数学运算并将运算得出的结果返回给这个变量**，

2、 因此对于这类运算通常有如下的快捷表达方式：

a = 2

**a = a \* 3**

同样也可写作：

a = 2

**a \*= 3**

3、   要注意到 变量 = 变量 运算 表达式 会演变成 变量 运算 = 表达式 。

* 1. 计算命令

1、 如果你有一个诸如**2 + 3 \* 4** 的表达式，是**优先完成加法还是优先完成乘法**呢？我们的高中数学知识会告诉我们应该先完成乘法。这意味着乘法运算符的优先级要高于加法运算符。

2、 下面将**给出 Python 中从最低优先级（最少绑定）到最高优先级（最多绑定）的优先级表**。这意味着，在给定的表达式中，**Python 将优先计算表中位列于后的较高优先级的运算符与表达式**。

3、 为了保持完整，下表是从 Python 参考手册 中引用而来。**你最好使用圆括号操作符来对运算**

**符与操作数进行分组，以更加明确地指定优先级。**这也能使得**程序更加可读**。你可以阅读改变运算顺序来了解更多的细节。

lambda ：Lambda 表达式

if - else ：条件表达式

or ：布尔“或”

and ：布尔“与”

not x ：布尔“非”

in, not in, is, is not, <, <=, >, >=, !=, == ：比较，包括成员资格测试

（Membership Tests）和身份测试（Identity Tests）。

| ：按位或

^ ：按位异或

& ：按位与

<<, >> ：移动

+, - ：加与减

\*, /, //, % ：乘、除、整除、取余

+x, -x, ~x ：正、负、按位取反

\*\* ：求幂

x[index], x[index:index], x(arguments...), x.attribute ：下标、切片、调用、属性引

用

(expressions...), [expressions...], {key: value...}, {expressions...} ：显示绑定或数

组、显示列表、显示字典、显示设置

我们还没有遇到的运算符将在后面的章节中加以解释。

在上表中位列同一行的运算符具有相同优先级。例如 + 和 - 就具有相同的优先级。

* 1. 改变运算顺序

1、 为了使**表达式更加易读，我们可以使用括号**。

2、 举个例子， 2 + (3 \* 4) 自是要比 2 + 3 \* 4 要更加容易理解，因为后者还要求你要了解运算符的优先级。

3、 和其它的一切一样，**使用括号同样也要适度（而不要过度）**，同时亦应不要像 (2 + (3 \* 4)) 这般冗余。

4、 使用**括号还有一个额外的优点——它能帮助我们改变运算的顺序**。同样举个例子，如果你希望在表达式中计算乘法之前应先计算加法，那么你可以将表达式写作 (2 + 3) \* 4 。

* 1. 关联性

1、 运算符通常**由左至右相互关联**。这意味着具有**相同优先级的运算符将从左至右**的方式依次进行求值。

2、 如 2 + 3 + 4 将会以 (2 + 3) +4 的形式加以计算。

* 1. 表达式

1、 案例（将其保存为 expression.py ）：

length = 5

breadth = 2

area = length \* breadth

print('Area is', area)

print('Perimeter is', 2 \* (length + breadth))

2、 输出：

$ python expression.py

Area is 10

Perimeter is 14

3、 它是如何工作的

         矩形的长度（Length）与宽度（Breadth）存储在以各自名称命名的变量中。我们使用它们并借助表达式来计算矩形的面积（Area）与周长（Perimeter）。我们将表达式 length \*breadth 的结果存储在变量 area 中并将其通过使用 print 函数打印出来。

在第二种情况中，我们直接在 print 函数中使用了表达式 2 \* (length + breadth) 的值。

同时，**你需要注意到 Python是如何漂亮地打印出 输出结果的。**

尽管我们没有特别在 Areais 和变量 area 之间指定空格，Python 会帮我们加上所以我们就能得到一个整洁的输出结果，同时程序也因为这样的处理方式而变得更加易读（因为我们**不需要在用以输出的字符串**

**中考虑空格问题**）。

这便是一个 Python 是如何让程序员的生活变得更加便捷美好的范例。

* 1. 总结

我们已经了解了如何使用**运算符、操作数与表达式**——这些是我们构建任何程序的基本块。

接下来，我们将看到如何在程序中善加利用这些语句。

1. 控制流

1、 截止到现在，在我们所看过的程序中，**总是有一系列语句从上到下精确排列，并交由 Python忠实地执行**。

2、 如果你**想改变这一工作流程，应该怎么做？**就像这样的情况：你需要程序作出一些决定，并依据不同的情况去完成不同的事情，例如依据每天时间的不同打印出 '早上好' 'Good Morning' 或 '晚上好' 'Good Evening'？

3、 正如你你可能已经猜测到的那番，这是通过**控制流语句来实现的**。

4、 在 Python 中**有三种控制流语句**—— if for 和 while 。

* 1. if 语句

1、**if 语句用以检查条件**：如果 **条件为真（True）**，我们将运行一块语句（称作 *if-block* 或 *if*块），**否则** 我们将运行另一块语句（称作 *else-block* 或 *else* 块）。其中 *else* 从句是可选的。

2、 案例（保存为 if.py ）：

number = 23

guess = int(input('Enter an integer : '))

if guess == number:

# 新块从这里开始

print('Congratulations, you guessed it.')

print('(but you do not win any prizes!)')

# 新块在这里结束

elif guess < number:

# 另一代码块

print('No, it is a little higher than that')

# 你可以在此做任何你希望在该代码块内进行的事情

else:

print('No, it is a little lower than that')

# 你必须通过猜测一个大于（>）设置数的数字来到达这里。

print('Done')

# 这最后一句语句将在

# if 语句执行完毕后执行。

3、 输出：

 $ python if.py

Enter an integer : 50

No, it is a little lower than that

Done

$ python if.py

Enter an integer : 22

No, it is a little higher than that

Done

$ python if.py

Enter an integer : 23

Congratulations, you guessed it.

(but you do not win any prizes!)

Done

4、它是如何工作的

         在这个程序中，我们根据用户猜测的数字来检查这一数字是否是我们所设置的。我们将变量number 设为任何我们所希望的整数，例如 23 。然后，我们**通过 input() 函数来获取用户的猜测数**。所谓函数是一种可重复使用的程序。我们将在下一章详细讨论它。

         我们为内置的**input 函数提供一串打印到屏幕上的字符串并等待用户的输入**。一旦我们输入了某些内容并按下键盘上的 enter 键， input() 函数将以字符串的形式返回我们所输入的内容。然后我们通过 int 将这个字符串转换成一个整数并将其储存在变量 guess 中。

         实际上，**int 是一个类（Class）**，但你现在你所需要知道的就是你可以使用它将一串字符串转换成一个整数（假设这个字符串的文本中含有一个有效的整数）。接下来，我们将用户提供的猜测数与我们所选择的数字进行对比。如果它们相等，我们就打印一条成功信息。

         在这里要注意到我们**使用缩进级别来告诉 Python 哪些语句分别属于哪个块**。这便是为什么在 Python 中缩进如此重要。我希望你能够坚持“缩进一致”的原则，你能做到吧？

         另外需要注意的是**if 语句在结尾处包含一个冒号**——我们借此向 Python 指定接下来会有一块语句在后头。

         然后，我们检查猜测数是否小于我们选择的数字，如果是，我们将告诉用户他们必须猜一个更高一些的数字。在这里我们使用的是 elif 语句，它们实际上将两个相连的 if else-if else 语句合并成一句 if-elif-else 语句。这能够使程序更加简便，并且可以减少所需要的缩进量。

           elif 和 else 同样都必须有一个冒号在其逻辑行的末尾，后面跟着与它们相应的语句块（当然，别忘了恰当的缩进）。

你可以在 if 块的 **一个 if 语句中设置另一个 if 语句**，并可以如此进行下去—**—这被称作嵌套的 if 语句。**

**要记住 elif 和 else 部分都是可选的**。一个最小规模且有效的 if 语句是这样的：

      if True:

      print('Yes, it is true')

**当 Python 完整执行了 if 语句及与其相关的 elif 和 else 子句后，它将会移动至包含 if 语句的代码块的下一句语句中。**在本例中，也就是主代码块（程序开始执行的地方），其下一句语句就是 print('Done') 语句。在完成这些工作后，Python 会发现已行至程序末尾并宣告工作的完成。

      尽管这是一个非常简单的程序，我也一直在其中指出你应该注意的事情。所有的这些都可算是简单易懂（对于那些具有 C/C++ 背景的人来说是相当简单易懂）。不过在开始时它们还是可能会不断吸引你的注意，不断地去在意它们。但**经过一些更丰富的操作后你就会习惯它们及其中的逻辑，它们对于你来说将会成为“自然而然”的事情**。

* 1. while 语句

1、 **while 语句能够让你在条件为真的前提下重复执行某块语句**。

2、 **while 语句**是 **循环（*Looping*） 语句的一种**。 **while 语句同样可以拥有 else 子句作为可选选项**。

3、 案例（保存为 while.py ）：

number = 23

running = True

while running:

guess = int(input('Enter an integer : '))

if guess == number:

print('Congratulations, you guessed it.')

# 这将导致 while 循环中止

running = False

elif guess < number:

print('No, it is a little higher than that.')

else:

print('No, it is a little lower than that.')

          else:

print('The while loop is over.')

         # 在这里你可以做你想做的任何事

print('Done')

4、 它是如何工作的

         在这一程序中，我们依旧通过**猜数游戏来演示**，不过新程序的优点在于能够允许用户持续猜测直至他猜中为止——而无需像我们在上一节中所做的那样，每次猜测都要重新运行程序。

         这种变化恰到好处地演示了 while 语句的作用。

         首先我们将 input 与 if 语句移到 while 循环之中，并在 while 循环开始前将变量running 设置为 True 。程序开始时，我们首先检查变量 running 是否为 True ，之后再执行相应的 *while* 块。在这一代码块被执行之后，将会重新对条件进行检查，在本例中也就是running 变量。如果它依旧为 True ，我们将再次执行 while 块，否则我们将继续执行可选

的 else 块，然后进入到下一个语句中。

**else 代码块在 while 循环的条件变为 False 时开始执行**——这个开始的时机甚至可能是在第一次检查条件的时候。**如果 while 循环中存在一个 else 代码块，它将总是被执行，除非你通过 break 语句来中断这一循环。**

**True 和 False 被称作布尔（Bollean）型**，你可以将它们分别等价地视为 1 与 0 。

* 1. for 循环

1、**for...in 语句**是另一种循环语句，其特点是会在一系列对象上进行迭代（*Iterates*），意即它会遍历序列中的每一个项目。

2、 我们将在后面的序列（Sequences）章节中了解有关它的更多内容。现在你所需要的就是**所谓队列就是一系列项目的有序集合**。

3、 案例（保存为 for.py ）：

for i in range(1, 5):

print(i)

else:

print('The for loop is over')

输出：

$ python for.py

1

2

3

4

The for loop is over

4、 在这一程序中，我们打印了一序列的数字。我们通过内置的 range 函数生成这一数字序列。

在这里我们所要做的事情是提供两个数字，而 range 将会返回一序列的数字，从第一个数字开始，至第二个数字结束。举个例子，**range(1,5) 将输出序列 [1, 2, 3, 4]** 。在默认情况下， range 将会以 1 逐步递增。如果我们向 range 提供第三个数字，则这个数字将成为逐步递增的加数。同样举个例子来说明，**range(1,5,2) 将会输出 [1, 3]** 。要记住这一序列扩展直到第二个数字，也就是说，它不会包括第二个数字在内。

另外需要注意的是，**range() 每次只会生成一个数字**，如果你希望获得**完整的数字列表**，要在使用**range() 时调用 list()** 。例如下面这样： list(range(5)) ，它将会返回 [0, 1, 2,3, 4] 。有关列表的详细解释将会在 《数据结构》一章呈现。

然后**for 循环就会在这一范围内展开递归**——**for i in range(1,5) 等价于 for i in [1,2, 3, 4]** ，**这个操作将依次将队列里的每个数字（或是对象）分配给 i** ，一次一个，然后以每个 i 的值执行语句块。在本例中，**我们这一语句块所做的就是打印出这些值**。

同样要记住，**else 部分是可选的。**当循环中包含他时，它总会**在 for 循环结束后开始执行，**除非程序遇到了 break 语句。

另一个需要注意的地方是**for...in 能在任何队列中工作**。在这里，我们有的是通过内置的range 函数生成的一串数字列表，但总体来说我们可以包含任何类型对象的队列！我们将会在后面的章节详细解释这一观念。

* 1. break 语句

1、**break 语句**用以**中断（Break）循环语句**，也就是**中止循环语句的执行**，即使循环条件没有变更为 False ，或队列中的项目尚未完全迭代依旧如此。

2、 有一点需要尤其注意，如果**你的 中断 了一个 for 或 while 循环**，**任何相应循环中的else 块都将不会被执行**。

3、 案例（保存为 break.py ）：

while True:

s = input('Enter something : ')

if s == 'quit':

break

print('Length of the string is', len(s))

print('Done')

         输出：

       $ python break.py

Enter something : Programming is fun

Length of the string is 18

Enter something : When the work is done

Length of the string is 21

Enter something : if you wanna make your work also fun:

Length of the string is 37

Enter something : use Python!

Length of the string is 11

Enter something : quit

Done

4、 它是如何工作的

         在本程序中，我们重复地接受用户的输入内容并打印出每一次输入内容的长度。我们通过**检查用户输入的是否是 quit 这一特殊条件来判断是否应该终止程序**。我们通过中断循环并转进至程序末尾来结束这一程序。

输入字符串的长度可以通过内置的 len 函数来找到。

记住， **break 语句同样可以适用于 for 循环**。

**Swaroop** 的诗意 **Python**

我所使用的输入内容是一首我所写的小诗：

Programming is fun

When the work is done

if you wanna make your work also fun:

use Python!

* 1. continue 语句

1、  continue 语句用以**告诉 Python 跳过当前循环块中的剩余语句**，并**继续该循环的下一次迭代**。

2、 案例（保存为 continue.py ）：

while True:

s = input('Enter something : ')

if s == 'quit':

break

if len(s) < 3:

print('Too small')

continue

print('Input is of sufficient length')

# 自此处起继续进行其它任何处理

输出：

$ python continue.py

Enter something : a

Too small

Enter something : 12

Too small

Enter something : abc

Input is of sufficient length

Enter something : quit

3、 它是如何工作的

         在本程序中，我们接受来自用户的输入内容，但是只有在输入的字符串其长至少 3 字符我们才会对其进行处理。为此，我们使用内置的 len 函数和来获取字符串的长度，如果其长度小于 3，我们便通过使用 continue 语句跳过代码块中的其余语句。否则，循环中的剩余语句将被执行，并在此处进行我们所希望的任何类型的处理。

         要**注意 continue 语句同样能用于 for 循环**。

* 1. 总结

我们已经了解了三种控制流语句——**if ， while 和 for ——及其相关的 break 与 continue 语句是如何工作的**。这些语句是 Python 中一些**最常用的部分**，因此，习惯去使用它们是必要的。

接下来，我们将了解如何创建并使用函数。

1. 函数

1、**函数**（Functions）是指可重复使用的程序片段。

2、 它们允许你为**某个代码块赋予名字**，允许你**通过这一特殊的名字在你的程序任何地方来运行代码块**，并可重复任何次数。这就是所谓的**调用（*Calling*）函数**。我们已经使用过了许多**内置的函数**，例如 len 和 range 。

3、 函数概念可能是在任何不平凡的软件（无论使用的是何种编程语言）中**最重要的构建块**，所以我们接下来将在本章中探讨有关函数的各个方面。

4、 **函数**可以通过**关键字 def 来定义**。这一关键字后跟一个**函数的标识符名称**，再跟**一对圆括号**，其中可以包括一些变量的名称，再**以冒号结尾**，结束这一行。随后而来的**语句块是函数的一部分**。下面的案例将会展示出这其实非常简单：

5、 案例（保存为 functon1.py ）：

def say\_hello():

# 该块属于这一函数

print('hello world')

# 函数结束

say\_hello() # 调用函数

say\_hello() # 再次调用函数

输出：

$ python function1.py

hello world

hello world

6、 它是如何工作的

         我们以上文解释过的方式定义**名为 say\_hello 的函数**。这个函数**不使用参数**，因此在括号中**没有声明变量**。函数的**参数**只是输入到函数之中，以便我可以传递不同的值给它，并获得相应的结果。要注意到我们可以两次调用相同的函数，这意味着我们不必重新把代码再写一次。

* 1. 函数参数

1、 **函数可以获取参数**，这个参数的值由你所提供，借此，函数便可以利用这些值来做一些事情。这些参数与变量类似，这些变量的值在我们**调用函数时已被定义**，且在函数运行时均已赋值完成。

2、 函数中的**参数**通过将其放置在用以定义函数的一对**圆括号中指定**，并通过**逗号予以分隔**。当我们**调用**函数时，我们以同样的形式提供需要的值。要注意在此使用的术语——在**定义函数**时给定的名称称作***“*形参*”***（*Parameters*），在**调用函数时**你所提供给函数的值称作***“*实参*”***（*Arguments*）。

3、案例（保存为 function\_param.py ）：

def print\_max(a, b):

if a > b:

print(a, 'is maximum')

elif a == b:

print(a, 'is equal to', b)

else:

print(b, 'is maximum')

# 直接传递字面值

print\_max(3, 4)

x = 5

y = 7

# 以参数的形式传递变量

print\_max(x, y)

输出：

$ python function\_param.py

4 is maximum

7 is maximum

4、 它是如何工作的

       在这里，我们将函数**命名为 print\_max** 并使用两个参数分别称作 a 和 b 。我们**使用一个简单的 if...else 语句来找出更大的那个数，并将它打印出来**。第一次调用函数 print\_max 时，我们以实参的形式直接向函数提供这一数字。在第二次调用时，我们将变量作为实参来调用函数。 print\_max(x, y) 将使得实参 x 的值将被赋值给形参a ，而实参 y 的值将被赋值给形参 b 。在两次调用中， print\_max 都以相同的方式工作。

* 1. 局部变量

1、 当你在一个函数的定义中声明**变量**时，它们不会以任何方式与身处函数之外但具有相同名称的变量产生关系，也就是说，这些变量名只存在于函数这一局部（*Local*）。这被称为变量的作用域（*Scope*）。所有变量的作用域是它们被定义的块，**从定义它们的名字的定义点开始**。

2、 案例（保存为 function\_local.py ）：

x = 50

def func(x):

print('x is', x)

x = 2

print('Changed local x to', x)

func(x)

print('x is still', x)

输出：

$ python function\_local.py

x is 50

Changed local x to 2

x is still 50

3、 它是如何工作的

         当我们第一次打印出存在于函数块的第一行的名为 x 的值时，Python 使用的是在函数声明之上的主代码块中声明的这一参数的值。

接着，我们将值 2 赋值给 x 。**x 是我们这一函数的局部变量**。因此，当我们改变函数中x 的值的时候，**主代码块中的 x 则不会受到影响**。

随着最后一句 print 语句，我们展示出主代码块中定义的 x 的值，由此确认它实际上不受先前调用的函数中的局部变量的影响。

* 1. global 语句

1、如果你想**给一个在程序顶层的变量赋值**（也就是说它不存在于任何作用域中，无论是函数还是类），那么你必须**告诉 Python 这一变量并非局部的，而是全局（*Global*）的**。

2、 我们需要**通过 global 语句**来完成这件事。因为在**不使用 global 语句**的情况下，**不可能为一个定义于函数之外的变量赋值**。

3、 你**可以使用定义于函数之外的变量的值**（假设函数中没有具有相同名字的变量）。然而，**这种方式不会受到鼓励而且应该避免，因为它对于程序的读者来说是含糊不清的，无法弄清楚变量的定义究竟在哪**。而**通过使用 global 语句便可清楚看出这一变量是在最外边的代码块中定义的**。

4、 案例（保存为 function\_global.py ）：

x = 50

def func():

global x

print('x is', x)

x = 2

print('Changed global x to', x)

func()

print('Value of x is', x)

输出：

$ python function\_global.py

x is 50

Changed global x to 2

Value of x is 2

5、 它是如何工作的

           global 语句用以**声明 x 是一个全局变量**——因此，当我们在函数中**为 x 进行赋值时**，这一改动**将影响到我们在主代码块中使用的 x 的值**。

         你可以在**同一句 global 语句中指定不止一个的全局变量**，例如 global x, y, z 。

* 1. 默认参数值

1、 对于一些函数来说，你可能为希望**使一些参数可选并使用默认的值**，以避免用户不想为他们提供值的情况。默认参数值可以有效帮助解决这一情况。你可以通过在函数定义时**附加一个赋值运算符（ = ）来为参数指定默认参数值**。

2、 要注意到，默认参数值应该是常数。更确切地说，默认参数值应该是不可变的——这将在后面的章节中予以更详细的解释。就目前来说，只要记住就行了。

3、 案例（保存为 function\_default.py ）：

def say(message, times=1):

print(message \* times)

say('Hello')

say('World', 5)

输出：

$ python function\_default.py

Hello

WorldWorldWorldWorldWorld

4、 它是如何工作的

         名为 say 的函数用以按照给定的次数打印一串字符串。如果我们**没有提供一个数值，则将按照默认设置**，只打印一次字符串。我们通过为参数 times 指定默认参数值 1 来实现这一点。

         在第一次使用 say 时，我们只提供字符串因而函数只会将这个字符串打印一次。在第二次使用 say 时，我们既提供了字符串，同时也**提供了一个参数 5** ，声明我们希望说（*Say*）这个字符串**五次**。

5、 注意

         只有那些位于参数列表末尾的参数才能被赋予默认参数值，意即在函数的参数列表中拥有默认参数值的参数不能位于没有默认参数值的参数之前。

这是因为值是按参数所处的位置依次分配的。举例来说， def func(a, b=5) 是有效的，但 def func(a=5, b) 是无效的。

* 1. 关键字参数

1、 **如果你有一些具有许多参数的函数，而你又希望只对其中的一些进行指定**，那么你可以**通过命名它们来给这些参数赋值——这就是关键字参数**（*Keyword Arguments*）——我们使用命名（关键字）而非位置（一直以来我们所使用的方式）**来指定函数中的参数**。

2、 这样做有**两大优点**——

         1）其一，我们**不再需要考虑参数的顺序**，函数的使用将更加容易。

         2）其二，我们可以**只对那些我们希望赋予的参数以赋值**，只要其它的参数都具有默认参数值。

3、 案例（保存为 `function\_keyword.py）：

def func(a, b=5, c=10):

print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)

func(25, c=24)

func(c=50, a=100)

输出：

$ python function\_keyword.py

a is 3 and b is 7 and c is 10

a is 25 and b is 5 and c is 24

a is 100 and b is 5 and c is 50

4、 它是如何工作的

         名为 func 的函数**有一个没有默认参数值的参数，**后跟**两个**各自带有**默认参数值**的参数。

         在**第一次调用**函数时， func(3, 7) ，参数 a 获得了值 3 ，参数 b 获得了值 7 ，而 c获得了默认参数值 10 。

         在**第二次调用**函数时， func(25, c=24) ，由于其所处的位置，变量 a 首先获得了值 25。然后，由于命名——即关键字参数——指定，变量 c 获得了值 24 。变量 b 获得默认参数值5 。

         在**第三次调用函**数时， func(c=50, a=100) ，我们全部使用关键字参数来指定值。在这里要注意到，尽管 a 在 c 之前定义，但我们还是我们在变量 a 之前指定了变量 c 。

* 1. 可变参数

1、 有时你可能想定义的**函数里面能够有任意数量的变量**，也就是**参数数量是可变**的，这可以通过**使用星号**来实现（将下方案例保存为 function\_varargs.py ）：

def total(a=5, \*numbers, \*\*phonebook):

print('a', a)

#遍历元组中的所有项目

for single\_item in numbers:

print('single\_item', single\_item)

#遍历字典中的所有项目

for first\_part, second\_part in phonebook.items():

print(first\_part,second\_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))

输出：

$ python function\_varargs.py

a 10

single\_item 1

single\_item 2

single\_item 3

Inge 1560

John 2231

Jack 1123

2、 它是如何工作的

         当我们声明一个诸如 \*param 的星号参数时，从此处开始直到结束的所有位置参数（Positional Arguments）都将被收集并汇集成一个称为“param”的元组（Tuple）。

         类似地，当我们声明一个诸如 \*\*param 的双星号参数时，从此处开始直至结束的所有关键字参数都将被收集并汇集成一个名为 param 的字典（Dictionary）。我们将在后面的章节探索有关元组与字典的更多内容。

* 1. return 语句

1、**return 语句**用于从**函数中返回**，也就是中断函数。我们也可以选择在中断函数时从函数中返回一个值。

2、 案例（保存为 function\_return.py ）：

def maximum(x, y):

if x > y:

return x

elif x == y:

return 'The numbers are equal'

else:

return y

print(maximum(2, 3))

输出：

$ python function\_return.py

3

3、 它是如何工作的

           maximum 函数将会**返回参数中的最大值**，在本例中是提供给函数的数值。它使用一套简单的if...else 语句来找到较大的那个值并将其返回。

         要注意到**如果 return 语句没有搭配任何一个值则代表着 返回 None 。** None 在 Python 中**一个特殊的类型，代表着虚无**。举个例子， 它用于指示一个变量没有值，如果有值则它的值便是 None（虚无） 。

**每一个函数都在其末尾隐含了一句 return None** ，除非你写了你自己的 return 语句。你可以运行 print(some\_function()) ，其中 some\_function 函数不使用 return 语句，就像这样：

def some\_function():

pass

**Python 中的 pass 语句用于指示一个没有内容的语句块**。

4、 提示：有一个名为 max 的内置函数已经实现了“找到最大数”这一功能，所以尽可能地使用这一内置函数。

* 1. DocStrings

1、 Python 有一个甚是优美的功能称作**文档字符串**（*Documentation Strings*），在称呼它时通常会使用另一个短一些的名字*docstrings*。**DocStrings 是一款你应当使用的重要工具**，它能够帮助你更好地记录程序并让其更加易于理解。令人惊叹的是，当程序实际运行时，我们甚至可以通过一个函数来获取文档！

2、 案例（保存为 function\_docstring.py ）：

def print\_max(x, y):

'''Prints the maximum of two numbers.打印两个数值中的最大数。

The two values must be integers.这两个数都应该是整数'''

# 如果可能，将其转换至整数类型

x = int(x)

y = int(y)

if x > y:

print(x, 'is maximum')

else:

print(y, 'is maximum')

print\_max(3, 5)

print(print\_max.\_\_doc\_\_)

输出：

$ python function\_docstring.py

5 is maximum

Prints the maximum of two numbers.

The two values must be integers.

3、 它是如何工作的

         函数的**第一行逻辑行中的字符串**是该**函数的 文档字符串（*DocString*）**。这里要注意文档字符串也适用于后面相关章节将提到**的模块（Modules）与类（Class）** 。

         该文档字符串所约定的是**一串多行字符串，其中第一行以某一大写字母开始，以句号结束。**第二行为空行，后跟的第三行开始是任何详细的解释说明。**在此强烈建议你在有关你所有非凡功能的文档字符串中都遵循这一约定。**

         我们可以通过使用函数的 \_\_doc\_\_ （注意其中的双下划綫）属性（属于函数的名称）来**获取函数 print\_max 的文档字符串属性**。**只消记住 Python 将所有东西都视为一个对象，这其中自然包括函数。**我们将在后面的类（Class）章节讨论有关对象的更多细节。

         如果你曾使用过 Python 的 help() 函数，那么你应该已经了解了文档字符串的用途了。它所做的便是获取函数\_\_doc\_\_ 属性并以一种整洁的方式将其呈现给你。你可以在上方的函数中尝试一下——只需在程序中包含 help(print\_max) 就行了。要记住你可以通过按下 q 键来退出 help 。

**自动化工具可以以这种方式检索你的程序中的文档**。因此，我强烈推荐你为你编写的所有不平凡的函数配以文档字符串。你的 Python 发行版中附带的**pydoc 命令与 help() 使用文档字符串的方式类似**。

* 1. 总结

我们已经了解了许多方面的函数，但我们依旧还未覆盖到所有类型的函数。不过，我们已经**覆盖到了大部分你每天日常使用都会使用到的 Python 函数**。

1. 模块

1、 在上一章，你已经了解了如何在你的程序中通过定义一次函数工作来重用代码。那么如果你想在你所**编写的别的程序中重用一些函数**的话，应该怎么办？正如你可能想象到的那样，答案是**模块**（Modules）。

2、**编写模块**有很多种方法，其中最简单的一种便是**创建一个包含函数与变量、以 .py 为后缀的文件**。

3、 另一种方法是使用**撰写 Python 解释器本身的本地语言来编写模块**。举例来说，你可以使用 C语言来撰写 Python 模块，并且在编译后，你可以通过标准 Python 解释器在你的 Python 代码中使用它们。

4、 **一个模块可以被其它程序导入并运用其功能**。我们在使用 Python 标准库的功能时也同样如此。首先，我们要了解如何使用标准库模块。

5、 案例 (保存为 module\_using\_sys.py ):

import sys

print('The command line arguments are:')

for i in sys.argv:

print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')

输出：

$ python module\_using\_sys.py we are arguments

The command line arguments are:

module\_using\_sys.py

we

are

arguments

The PYTHONPATH is ['/tmp/py',

# many entries here, not shown here

'/Library/Python/2.7/site-packages',

'/usr/local/lib/python2.7/site-packages']

6、 它是如何工作的

         首先，我们**通过 import 语句导入 sys 模块**。基本上，这句代码将转化为我们告诉 Python我们希望使用这一模块。 **sys 模块**包含了与 Python 解释器及其环境相关的功能，也就是所谓的**系统功能**（*sys*tem）。

         当 Python **运行 import sys 这一语句时**，它会开始寻找 sys 模块。在这一案例中，由于其是一个内置模块，因此 Python 知道应该在哪里找到它。

         如果它**不是一个已编译好的模块**，即用 Python 编写的模块，那么 Python 解释器将从它的sys.path 变量所提供的目录中进行搜索。如果找到了对应模块，则该模块中的语句将在开始运行，并能够为你所使用。在这里需要注意的是，初始化工作只需在我们第一次导入模块时完成。

**sys 模块中的 argv 变量通过使用点号予以指明**，也就是 sys.argv 这样的形式。它清晰地表明了这一名称是 sys 模块的一部分。这一处理方式的另一个优点是这个名称不会与你程序中的其它任何一个 argv 变量冲突。

**sys.argv 变量**是一系列字符串的列表（*List*）（列表将在后面的章节予以详细解释）。具体而言， sys.argv 包含了命令行参数（*Command Line Arguments*）这一列表，也就是使用命令行传递给你的程序的参数。

**如果你正在使用一款 IDE 来编写并运行这些程序，请在程序菜单中寻找相关指定命令行参数的选项。**

         在这里，当我们运行 python module\_using\_sys.py we are arguments 时，我们通过 python 命令来运行 module\_using\_sys.py 模块，后面的内容则是传递给程序的参数。 Python 将命令行参数存储在 sys.argv 变量中供我们使用。

         在这里要记住的是，运行的脚本名称在 sys.argv 的列表中总会位列第一。因此，**在这一案例中我们将会有如下对应关系： 'module\_using\_sys.py' 对应 sys.argv[0] ， 'we' 对应sys.argv[1] ， 'are' 对应 sys.argv[2] ， 'arguments' 对应 sys.argv[3] 。**要注意到

Python 从 0 开始计数，而不是 1。

           sys.path 内包含了导入模块的字典名称列表。你能观察到 sys.path 的第一段字符串是空的——这一空字符串代表当前目录也是 sys.path 的一部分，它与 PYTHONPATH 环境变量等同。这意味着你可以直接导入位于当前目录的模块。否则，你必须将你的模块放置在sys.path 内所列出的目录中。

         另外要注意的是**当前目录指的是程序启动的目录**。你可以通过运行 import os;print(os.getcwd()) 来查看你的程序目前所处在的目录。