

目 录

致谢
[README](#)
前言
简介
安装
基础数据类型
列表
模块和函数
迭代与循环
字典
类
迭代器
生成器
协程
异步编程

致谢

当前文档《Full Speed Python(进击的Python)中文版》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-06-02。

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，如果您在日常生活、工作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/full-speed-python-chinese>

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。

README

- [Full-speed-python](#) 的个人翻译版本
- [目录](#)
- [来源\(书栈小编注\)](#)

Full-speed-python 的个人翻译版本

原书地址: <https://github.com/joaovventura/full-speed-python>

进入GitBook在线阅读: [GitBook](#)

进入Love2io在线阅读: [@hubertroy/doc/full-speed-python-chinese">Love2io](#)

目录

- 简介
- 安装
- 基础数据类型
- 列表
- 函数
- 循环
- 字典
- 类
- 迭代器
- 生成器
- 协程
- 异步

来源(书栈小编注)

<https://github.com/HuberTRoy/full-speed-python-chinese>

前言

偶然发现的一篇Python的教程，内容从基础的下载安装到异步协程都有涵盖。
每节又配有几个小习题供练手，觉得十分不错，随手翻译下来~。

有错误烦请移步至[GitHub](#)发起Issue。

感谢~。

简介

- [简介](#)

简介

本书的目的是以实际例子来教会大家Python这门编程语言。方法很简单：每一个主题都有一个简短的介绍，读者通过解决实际的问题来学的更多知识。

这些实例被广泛用在我在Superior科技学院的web开发与分布式计算课程中。通过这些实例，学生们可以在一个月内快速掌握Python。实际上，曾在第二学年学习过软件工程的学生，可以在二周内熟悉Python的语法，并在第三周使用sockets写出分布式计算网络的客户端。

请注意，本书仍在不断创作，可能会包含一些拼写错误。但对于想使用它来进行学习的人来说，这应该不会成为一个很大的困难。同时，真心祝愿你能从本书中获益。

本书中的源码发布在github上。同时欢迎各位发起PR来纠正拼写错误，发布新的练习项目，对目前已存在的项目提供更详细的说明等。

学习的路上一路顺风~。

安装

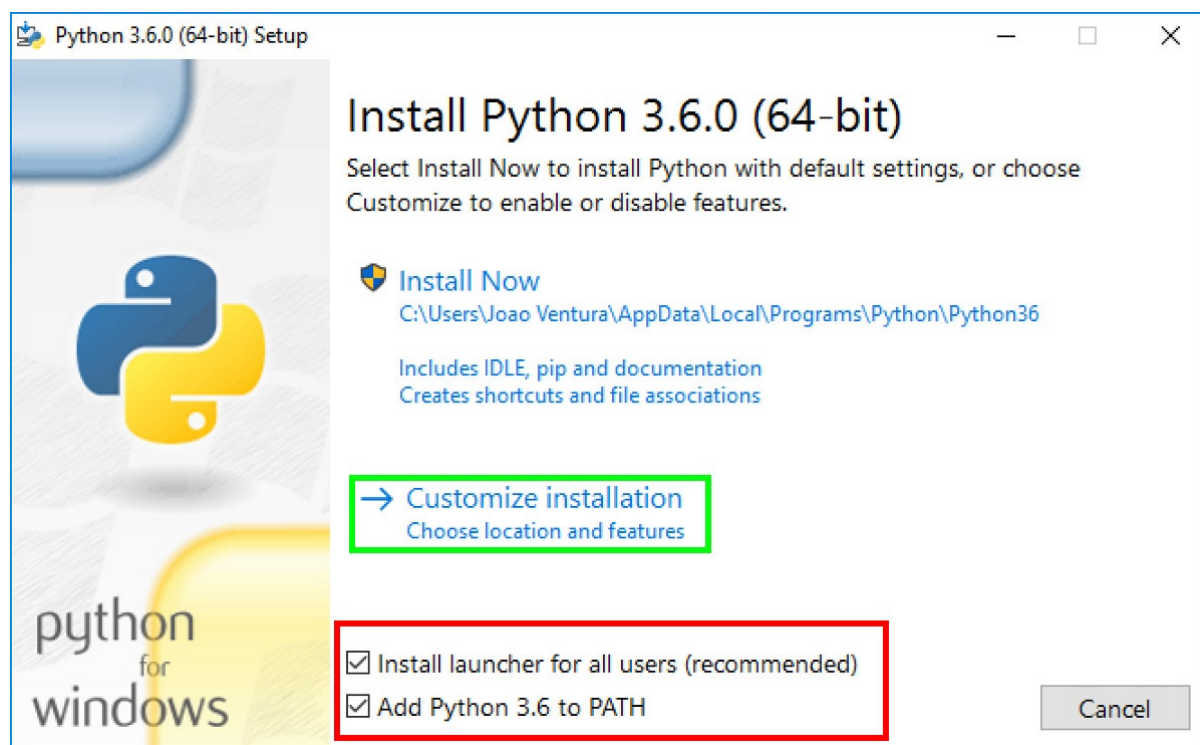
- 安装
 - 在Windows上安装
 - 在macOS上安装
 - 在Linux上安装

安装

本节我们将会安装Python到我们的本地电脑中。

在Windows上安装

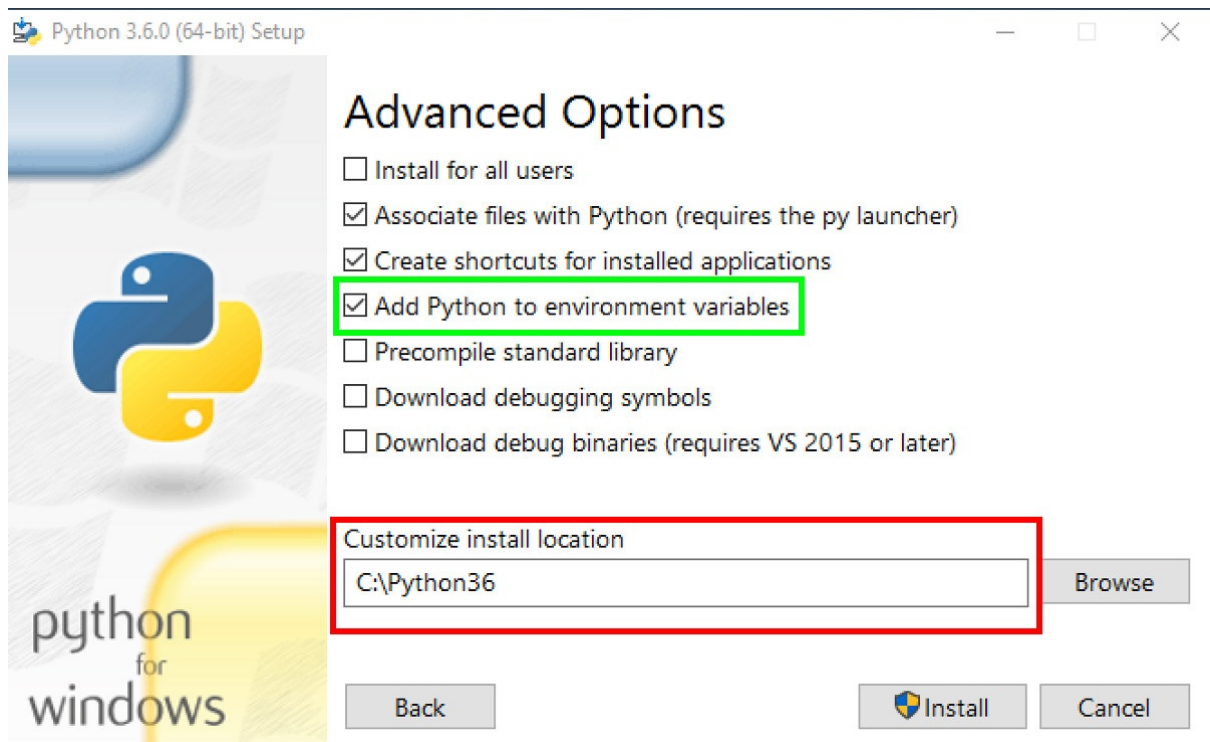
1. 首先从 <https://www.python.org/downloads/windows/> 上选择合适的Python 3版本下载并执行安装。写作本书时最新的版本是3.6.4。
2. 确保勾选了“Install launcher for all users (为所有用户执行安装)”和“Add Python to PATH (将Python添加到环境目录中)”并选择“Customize installation (自定义安装)”。



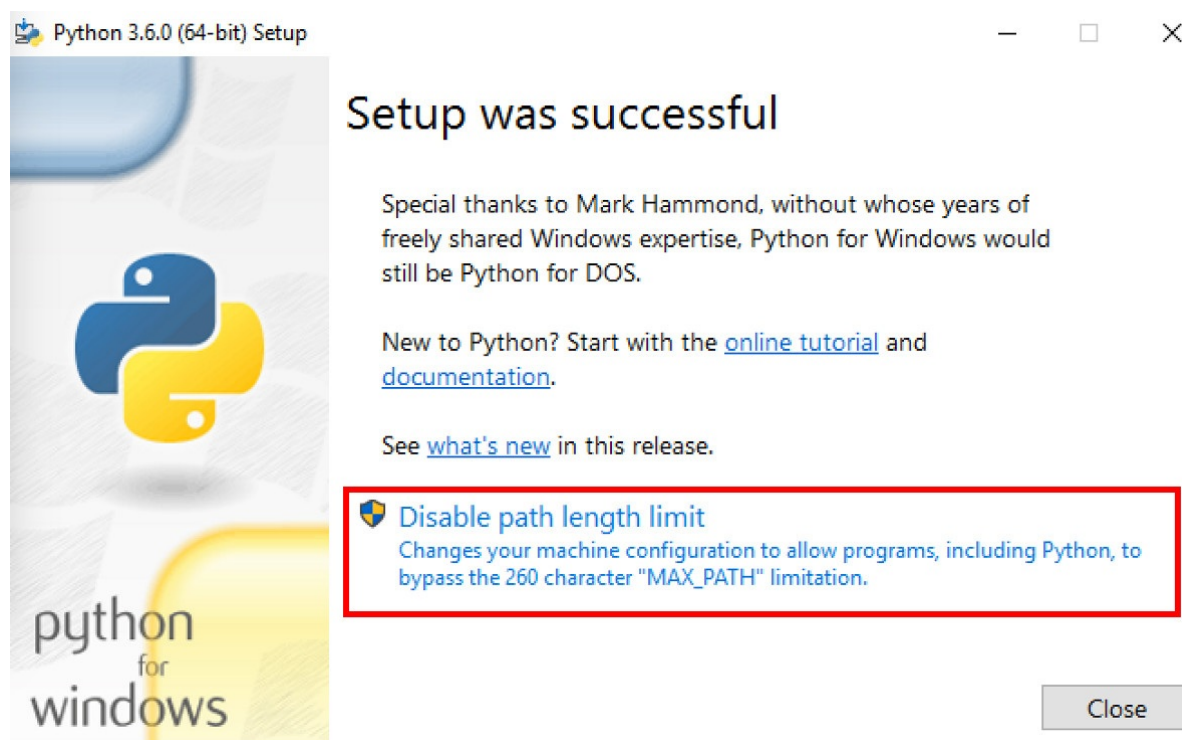
1. 在下一个视图“Optional Features”中选择需要添加的东西，不过请确保“pip”和“Pylauncher (for all users)”是勾选的。pip是Python的包管理器，可以让

你很方便地安装Python包。

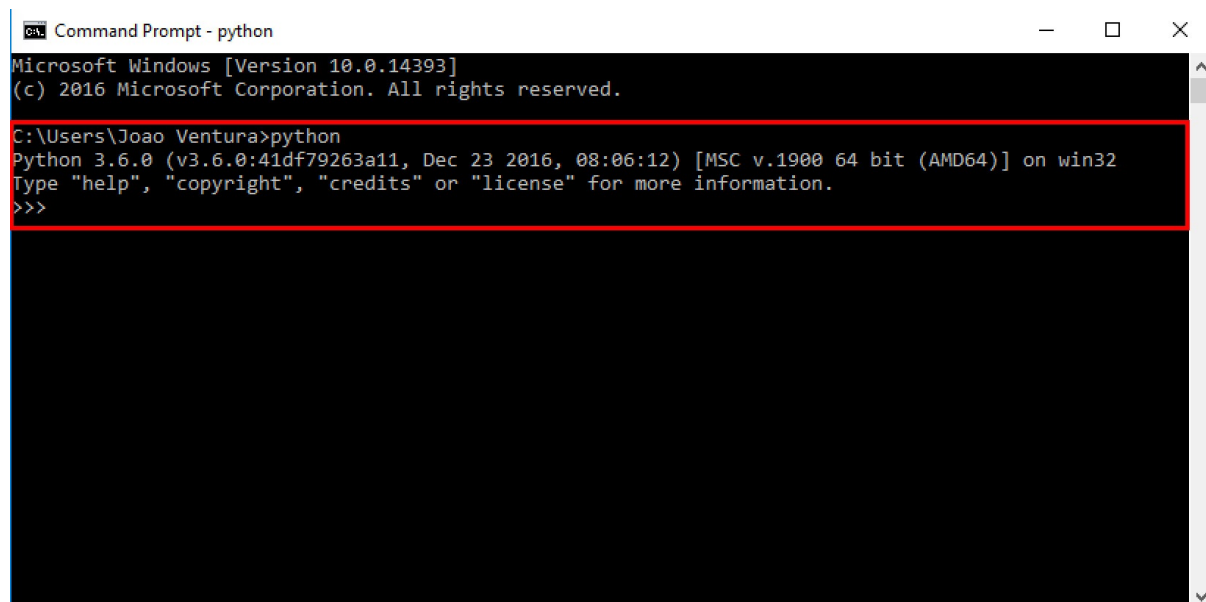
2. 在接下来的“Advanced Options”中确保勾选了“Add Python to environment variables”。同时我建议你修改一下安装目录，比如使用“C:\Python36\”，这样在需要时可以很方便地找到Python的安装目录。



1. 最后，允许Python在文件系统上可以超过260个字符，这一条要选择“Disable path length limit (禁用路径长度限制)”并关闭安装对话框。



1. 现在，打开命令行（cmd）并输出 `python` 或 `python3`。如果安装的过程顺利，那你应该会看到 `Python REPL`。`REPL`（意思是读取，评估，打印，循环）是一个允许你编写一些小型Python编码的环境。最后使用 `exit()` 关掉 `REPL`。

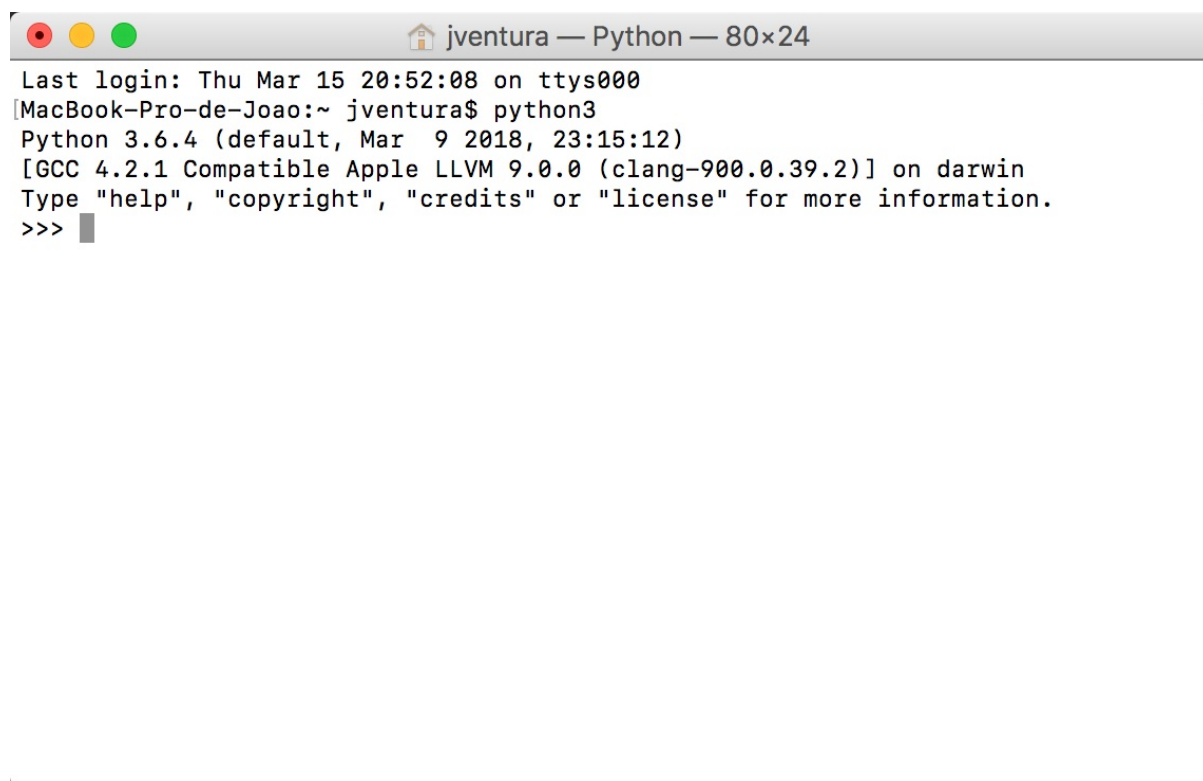


在macOS上安装

你可以从 <https://www.python.org/downloads/mac-osx/> 上选择合适的安装包下载。确保你下载的是最新的Python 3(写作时最新版本是3.6.4)。同时你可以使用

Homebrew (<https://brew.sh/>) 来进行下载。

使用Homebrew只需要在终端输入 `brew install python3` 即可。你也可以使用MacPorts包管理器 (<https://www.macports.org/>) 进行下载，所需要的命令是 `port install python36`。

A terminal window titled 'jventura — Python — 80x24'. The output shows the last login time, the command 'python3' being executed, and the resulting Python 3.6.4 environment information, including the GCC version and the system (darwin). The prompt '>>>' is visible at the end of the output.

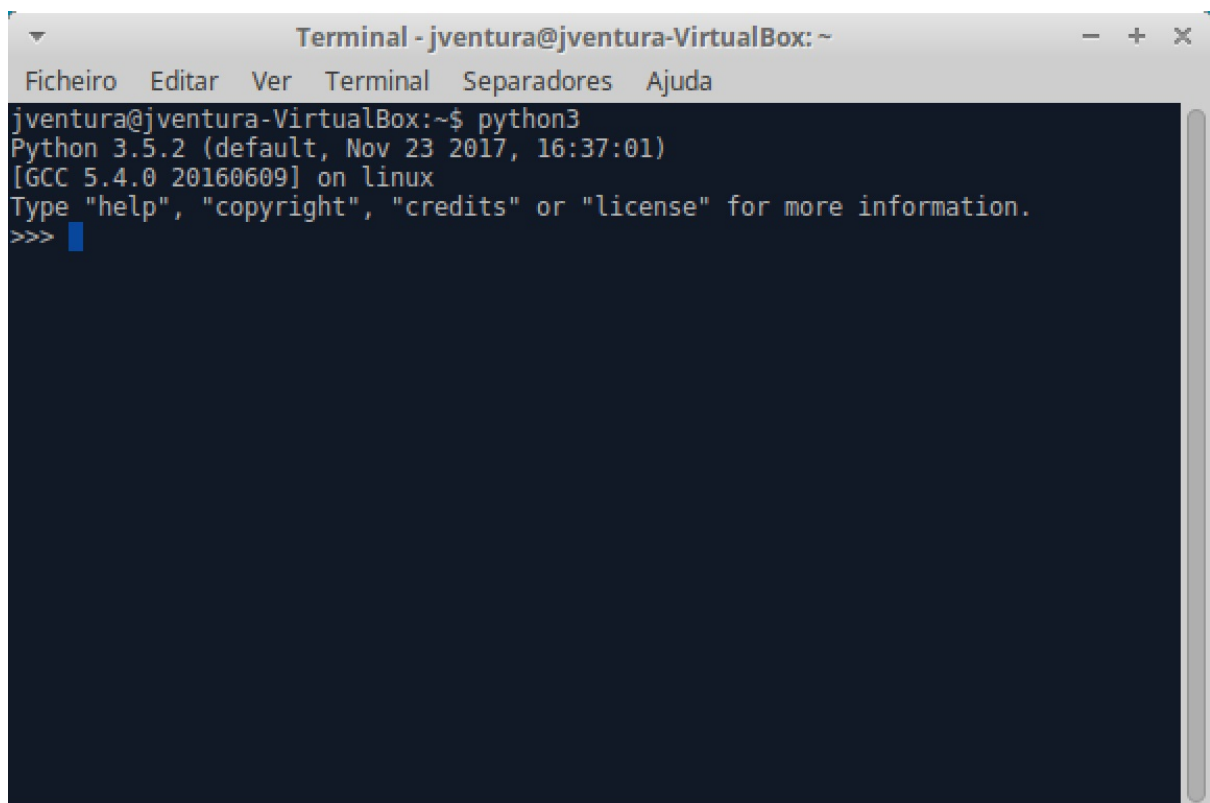
```
Last login: Thu Mar 15 20:52:08 on ttys000
[MacBook-Pro-de-Joao:~ jventura$ python3
Python 3.6.4 (default, Mar  9 2018, 23:15:12)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

最后，打开终端并输入 `python3` 你应该会看到 `Python REPL`。按下 `Ctrl + D` 或输入 `exit()` 可以退出。

在Linux上安装

在Linux上安装Python，你可以在 <https://www.python.org/downloads/source/> 中下载最新的Python 3 源码，或者使用包管理器（apt-get, aptitude, synaptic或者其他的）来安装。之后确保你已经成功安装了Python 3，你可以运行 `python3 --version` 来查看是否已经安装成功。

最后在终端中输入 `python3` 你就可以看到Python REPL出现了。按下 `Ctrl + D` 或输入 `exit()` 可以退出。

A terminal window titled "Terminal - jventura@jventura-VirtualBox: ~" with a menu bar containing "Ficheiro", "Editar", "Ver", "Terminal", "Separadores", and "Ajuda". The terminal output shows the command "python3" being executed, resulting in the following text: "Python 3.5.2 (default, Nov 23 2017, 16:37:01)", "[GCC 5.4.0 20160609] on linux", and "Type 'help', 'copyright', 'credits' or 'license' for more information.". The prompt ">>>" is followed by a blue cursor.

```
Terminal - jventura@jventura-VirtualBox: ~
Ficheiro  Editar  Ver  Terminal  Separadores  Ajuda
jventura@jventura-VirtualBox:~$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

基础数据类型

- [数字和字符串](#)
 - [数字部分练习](#)
 - [字符串部分练习](#)

数字和字符串

本节我们将了解最基础的数据类型，数字和字符串。打开你的 [Python REPL](#) 并写出以下语句。

```
1. >>> a = 2
2. >>> type(a)
3. <class 'int'>
4. >>> b = 2.5
5. >>> type(b)
6. <class 'float'>
```

这样你就定义了两个变量（“a”和“b”）它们保存了一些数字：变量“a”保存的是一个整数，而“b”保存的是一个实数。

我们现在可以使用刚才定义的两个变量或者其他数字来做些计算：

```
1. >>> a + b
2. 4.5
3. >>> (a + b) * 2
4. 9.0
5. >>> 2 + 2 + 4 - 2/3
6. 7.333333333333333
```

Python还支持字符串类型。字符串是一些连续的字符（比如一个单词），可以使用单引号或双引号来定义：

```
1. >>> hi = "hello"
2. >>> hi
3. 'hello'
4. >>> bye = 'goodbye'
5. >>> bye
6. 'goodbye'
```

你可以将两个字符串相加来连接它们，但你不能将两个不同的数据类型相加，比如一个字符串一个整数。

```
1. >>> hi + "world"
```

```

2. 'helloworld'
3. >>> "Hello" + 3
4. Traceback (most recent call last):
5. File "<stdin>", line 1, in <module>
6. TypeError: must be str, not int

```

不过你可以用乘来将原字符串翻倍：

```

1. >>> "Hello" * 3
2. 'HelloHelloHello'

```

数字部分练习

- 猜一下进行以下数学计算会发生什么： $((3 / 2))$, $((3 // 2))$, $((3 \% 2))$, $((3^{**}2))$
提示：阅读下 <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex> 了解更过情况。
- 计算下列数字序列的平均数：(2, 4), (4, 8, 9), (12, 14/6, 15)
- 一个球体的体积是 $(4/3 \pi r^3)$ 。请计算出一个半径为5的球体的体积。
提示：可以创建一个值为 `3.1415` 的变量 `pi`。
- 使用取模运算 (%) 来检查下列数字是单数还是双数：(1, 5, 20, 60/7)。

提示：当 (x) 是双数时，(x/2) 总会是0。

- 请找到合适的 (x) 和 (y) 可以让 $(x < 1/3 < y)$ 在Python REPL中返回True。
提示： $(0 < 1/3 < 1)$ 。

字符串部分练习

参阅Python 中关于字符串的文档

(<https://docs.python.org/3/library/stdtypes.html?#text-sequence-type-str>)，然后来解决下列问题：

- 初始化一个字符串为 `"abc"`，并将其赋值给变量 `"s"`：

- 使用一个函数获取出该字符串的长度。
- 使用一些操作符让 `"abc"` 变成 `"aaabbbcccc"`。
提示：使用字符串连接与索引。

- 初始化一个字符串为 `"aaabbbcccc"`，并将其赋值给变量 `"s"`：

- 使用一个函数获取出 `"b"` 第一次出现的位置，以及 `"ccc"` 第一次出现的位置。

ii. 使用一个函数替换所有的 `"a"` 为 `"X"`，然后使用相同的函数只替换第一个 `"a"` 为 `"X"`。

3. 给你一个字符串 `"aaa bbb ccc"`，使用什么操作可以让它们变成以下字符串？ 你可以使用 `replace` 函数。

- i. `"AAA BBB CCC"`
- ii. `"AAA bbb CCC"`

列表

- [列表](#)
 - [列表部分练习](#)
 - [列表推导式](#)
 - [列表推导式部分练习](#)

列表

Python中的列表是一个可存放一组元素的数据结构。列表中的元素可以是好几种类型，你可以在同一个列表中混用多种类型，尽管一般来说里面存放的应该都是同一种数据类型。

创建一个列表使用中括号来完成，每个元素中间使用逗号隔开。列表中的元素可以使用它们位置信息来访问，0是第一个。

```
1. >>> l = [1, 2, 3, 4, 5]
2. >>> l[0]
3. 1
4. >>> l[1]
5. 2
```

你能试着获取出数字4吗？

有时你想要的是列表中的一小段，一个子列表。子列表可以使用一种叫切片的方式获取到，使用切片需要同时定义开始和结束的索引。

```
1. >>> l = ['a', 'b', 'c', 'd', 'e']
2. >>> l[1:3]
3. ['b', 'c']
```

最后，列表同样支持算术操作，像是将两个列表合并到一起或者重复其中的元素。

```
1. >>> [1,2] + [3,4]
2. [1, 2, 3, 4]
3. >>> [1,2] * 2
4. [1, 2, 1, 2]
```

列表部分练习

创建一个名为 `I` 的变量，并输入以下值：（ [1, 4, 9, 10, 23] ）。

查阅Python文档

(<https://docs.python.org/3.5/tutorial/introduction.html#lists>) 中关于列表的部分来完成以下练习：

1. 使用列表切片获取出子列表 `[4, 9]` 和 `[10, 23]`。
2. 将 `90` 添加到列表 `l` 的末尾。尝试找一下合并两个列表和使用 `append` 方法有什么不同。
3. 计算出列表中所有元素的平均数。你可以使用 `sum` 和 `len` 两个函数。
4. 删除子列表 `[4, 9]`。

列表推导式

列表推导式是一种生成列表的简洁方式。在方括号中写入一个包含 `for` 关键字的表达式即可完成。生成的内容与表达式有关。

下面演示了以一个列表中的数生成另一个内容为其数字平方的列表。

```
1. >>> [x*x for x in [0, 1, 2, 3]]
2. [0, 1, 4, 9]
```

出于灵活性考虑，列表表达式一般与 `range` 函数连用：

```
1. >>> [x*x for x in range(4)]
2. [0, 1, 4, 9]
```

有时你想基于给定条件过滤一些元素。这时 `if` 关键字就上场了：

```
1. >>> [x for x in range(10) if x % 2 == 0]
2. [0, 2, 4, 6, 8]
```

上面的例子返回0-10中所有的双数。更多的例子请参阅

<https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions>

列表推导式部分练习

1. 使用列表推导式创建一个包含前10个数的平方的列表。
2. 使用列表推导式创建一个包含前20个数的立方的列表。
3. 使用列表推导式创建一个包含0-20间所有偶数的列表，创建另一个包含其中所有的奇数。
4. 创建一个包含0-20间所有偶数的平方的列表，使用 `sum` 函数将它们相加。结果应该是1140。
5. 创建一个包含0-20间所有偶数的平方，但忽略其中可以被3整除的列表。换句话说就是可以被2整除但不能被3整除。

请参阅Python文档中关于 `and` 关键字的用法。返回的列表应该是 `[4, 16, 64, 100, 196, 256]`。

模块和函数

- [模块和函数](#)
 - [函数部分练习](#)
 - [递归函数](#)
 - [递归函数的练习](#)

模块和函数

本节我们要讨论下模块和函数。

一个函数是一块用来执行单个操作的代码块。

一个模块则是一个Python文件，里面包含了变量，函数和其他的东西。

我们启动Python REPL然后使用下`math`模块，这个模块提供了一些数学方面的函数：

```
1. >>> import math
2. >>> math.cos(0.0)
3. 1.0
4. >>> math.radians(275)
5. 4.799655442984406
```

函数中包含了一组表达内容的序列，在调用的时候会被执行出来。

下面我们定义了一个 `do_hello` 函数，它的作用是打印两条信息：

```
1. >>> def do_hello():
2. ...     print("Hello")
3. ...     print("World")
4. ...
5. >>> do_hello()
6. Hello
7. World
```

确保你在函数中的`print`表达式前插入了一个`tab`。

在Python中`tab`和`space`是非常重要的，定义一个代码块或多或少都会依赖于这样的结构。

比如，`print`表达式被包含在 `do_hello` 函数里面时，就必须有一个`tab`。

函数还可以接受参数，并且也能返回一个值（使用`return`关键字来完成）。

```
1. >>> def add_one(val):
2. ...     print("Function got value", val)
3. ...     return val + 1
4. ...
```

```

5. >>> value = add_one(1)
6. Function got value 1
7. >>> value
8. 2

```

函数部分练习

1. 定义一个名为**add2**的函数，它接受两个数字作为参数，返回的值为这两个数字之和。之后定义一个名为**add3**的函数，它接受3个数字并且返回这三个数字的和。
2. 定义一个函数，返回接受的两个数字中较大的那个数字。你可以使用**if**关键字来比较两个数字：<https://docs.python.org/3/tutorial/controlflow.html#if-statements>。
3. 定义一个名为**is_divisible**的函数，它接受两个参数（“a”和“b”），如果“a”可以被“b”整除那么返回 `True` 否则返回 `False`。
提示： 一个数字可以被另一个数字整除时，它余下的部分为0。你可以使用取模操作符（%）。
4. 定义一个名为**average**的函数，计算出所传入的列表的平均值。你可以使用 `sum` 和 `len` 函数。

递归函数

在计算机编程中，递归函数的意思是调用自身。比如一个阶乘函数

```

1. f(x) =
2. { 1, if x = 0. }
3. { x × f(x - 1), otherwise. }

```

作为例子，我们传入5，它的计算过程是这样的：

```

1. 5! = 5 × 4!
2. = 5 × 4 × 3!
3. = 5 × 4 × 3 × 2!
4. = 5 × 4 × 3 ×

```

5的阶乘本质上是5乘4的阶乘...最终，是1（或0）的阶乘也就是1然后结束递归。在Python中，我们可以这样写：

```

1. def factorial(x):
2.     if x == 0:
3.         return 1
4.     else:

```

```
5.         return x * factorial(x-1)
```

递归的技巧是必须有一个基础条件，在这个条件下递归会终止，而递归的过程必须逐渐靠近这个基础条件。

阶乘这个例子中，我们知道数字为0时阶乘的结果是1，然后当阶乘的数大于0时则是之前数字的阶乘一直延续到0。

递归函数的练习

1. 写一个阶乘函数并测试几个不同的数据。同时使用计算器来验证其正确性。
2. 写一个递归函数，该函数会计算出(n)的前几个整数的和（n作为参数传入）。想想基础条件应该是什么（是不是加到0为止？），思考下递归例子是怎么做的。
3. 斐波那契数列是一个数字序列，它的每一个数字都是前2个数字之和。请根据以下信息，写出一个递归的斐波那契数列（fib(n)）。

```
1. fib(n) =  
2. { 0, if x = 0. }  
3. { 1, if x = 1. }  
4. { fib(n - 1) + fib(n - 2), otherwise }
```

检查下你最初的几个结果是否与结果对应了：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

迭代与循环

- [迭代与循环](#)
 - [for循环部分练习](#)
 - [while循环部分练习](#)

迭代与循环

本节我们探索的主题是迭代与循环。循环通常在计算机编程用于自动执行重复性任务。

在Python中最常用的迭代形式就是**for**循环了。**for**循环允许你迭代出列表中所有的项，迭代出来后你可以做任何你想做的事情。

比如，我们创建了一个列表，并打印出其中所有元素的平方。

```
1. >>> for value in [0, 1, 2, 3, 4, 5]:
2. ...     print(value * value)
3. ...
4. 0
5. 1
6. 4
7. 9
8. 16
9. 25
```

这样的形式简单又强大！**for**循环是许多编程中的基础。

比如，你已经知道的**sum(list)**这个函数，它会将列表中所有的元素相加，但你同样可以使用**for**循环来干这件事情：

```
1. >>> mylist = [1,5,7]
2. >>> sum = 0
3. >>> for value in mylist:
4. ...     sum = sum + value
5. ...
6. >>> print(sum)
7. 13
```

其本质是创建了一个变量“sum”，然后不断的将列表中的元素与其相加。

有时候，你可能不想使用列表中的值，而只需要它们的索引位置。比如，我们不止想获取出它的值还想获取出它在列表中的位置。

请看以下例子：

```
1. >>> mylist = [1,5,7]
```

```

2. >>> for i in range(len(mylist)):
3. ...     print("Index:", i, "Value:", mylist[i])
4. ...
5. Index: 0 Value: 1
6. Index: 1 Value: 5
7. Index: 2 Value: 7

```

你可以看到我们并没有迭代列表本身，而是迭代了这个列表长度的“range”。“range”函数返回的是一个特殊列表：

```

1. >>> list(range(3))
2. [0, 1, 2]

```

所以当使用“range”时迭代的就不是“mylist”而是一些用于访问列表里的值的数字索引。更多关于“range”的内容你可以参阅Python文档中关于“range”的部分。

<https://docs.python.org/3/tutorial/controlflow.html#the-range-function>

有时你可能同时需要这两个（索引和值），这时你可以使用**enumerate**函数：

```

1. >>> mylist = [1,5,7]
2. >>> for i, value in enumerate(mylist):
3. ...     print("Index:", i, "Value:", value)
4. ...
5. Index: 0 Value: 1
6. Index: 1 Value: 5
7. Index: 2 Value: 7

```

请记住Python列表中第一个值的索引总是0。

最后，Python里还提供**while**声明来让我们重复做一些事情，只要这个特定的条件是**True**就会一直执行。

比如，下面这个例子中*n*开始于10，条件是“*n*”大于0。每次都会从*n*中减去1。当*n*变为0时，“*n*>0”这个条件就失效了，此循环也就结束了。

```

1. >>> n = 10
2. >>> while n > 0:
3. ...     print(n)
4. ...     n = n-1
5. ...
6. 10
7. 9
8. 8
9. 7
10. 6
11. 5

```

```

12. 4
13. 3
14. 2
15. 1

```

这个循环永远也不会打印0。

for循环部分练习

解决这一部分你可能很想参阅Python文档中关于循环的部分：

<https://docs.python.org/3/tutorial/controlflow.html#for-statements>

1. 创建一个名为**add**的函数，它接受一个列表作为参数，返回的是列表中所有元素的和。请使用**for**循环来完成。
2. 创建一个函数，它接受一个列表作为参数，返回的是列表中最大的一个数。
提示：每迭代完一次你可能都要保存一下目前为止得到的最大的数以进行比较。
3. 修改之前的那个函数，现在它返回的是一个列表，列表中第一个元素为最大的一个值，第二个元素为这个值在原列表中的位置。
提示：除了保存目前为止最大的数，你还要保存下它出现的位置。
4. 实现一个函数，它返回所接受的列表的翻转列表。
提示：你可以创建一个空列表，然后从原列表中倒序的添加值。你可能会想参阅Python文档中关于列表的部分：
<https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>
5. 写一个名为**is_sorted**的函数，它接受一个列表作为参数，如果这个列表里的内容是以升序排列的话则返回**True**。
比如 `[1, 2, 2, 3]` 就是**True**， `[1, 2, 3, 2]` 则不是。
提示：如果你要将一个数字与接下来的数字做对比，你可能会用到索引，或者你也可以将其保存到一个变量中。
6. 写一个名为**is_sorted_dec**的函数，与之前的那个相似但所有的项都是以降序排列的。
7. 写一个名为**has_duplicates**的函数，验证所传入的列表里是否存在重复的数。
提示：你可能需要使用两个**for**循环，在一个**for**循环里在使用一个**for**来检查是否有重复的数。

while循环部分练习

1. 写一个函数，接受一个数字作为参数并且以降序形式打印出来，打印时标出该数字是奇数还是偶数，一直到0。

```
1. >>> even_odd(10)
2. Even number: 10
3. Odd number: 9
4. Even number: 8
5. Odd number: 7
6. Even number: 6
7. Odd number: 5
8. Even number: 4
9. Odd number: 3
10. Even number: 2
11. Odd number: 1
```

字典

- [字典](#)
 - [字典部分练习](#)
 - [子字典部分练习](#)

字典

本节我们将使用下Python中的字典。字典是一个数据结构，它的索引值是给定的key(是一个key-value对的形式)。

下面的例子展示了以同学的名字为索引，值为年龄的字典：

```
1. ages = {
2.     "Peter": 10,
3.     "Isabel": 11,
4.     "Anna": 9,
5.     "Thomas": 10,
6.     "Bob": 10,
7.     "Joseph": 11,
8.     "Maria": 12,
9.     "Gabriel": 10,
10. }
11. >>> print(ages["Peter"])
12. 10
```

使用一个字典的 `items` 可以迭代出其中的内容，比如这样：

```
1. >>> for name, age in ages.items():
2.     ... print(name, age)
3. ...
4. Peter 10
5. Isabel 11
6. Anna 9
7. Thomas 10
8. Bob 10
9. Joseph 11
10. Maria 12
11. Gabriel 10
```

不过，字典的key并不需要是一个字符串，它可以是任何不可变对象：

```
1. d = {
2.     0: [0, 0, 0],
```



```

3. 1: [1, 1, 1],
4. 2: [2, 2, 2],
5. }
6. >>> d[2]
7. [2, 2, 2]

```

同时你可以将另一个字典作为某个值写入：

```

1. students = {
2. "Peter": {"age": 10, "address": "Lisbon"},
3. "Isabel": {"age": 11, "address": "Sesimbra"},
4. "Anna": {"age": 9, "address": "Lisbon"},
5. }
6. >>> students['Peter']
7. {'age': 10, 'address': 'Lisbon'}
8. >>> students['Peter']['address']
9. 'Lisbon'

```

因此，这种结构非常容易书写阶级式内容。

字典部分练习

自行研究Python文档 <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict> 来解决下列练习。

先定义一个Python字典：

```

1. ages = {
2. "Peter": 10,
3. "Isabel": 11,
4. "Anna": 9,
5. "Thomas": 10,
6. "Bob": 10,
7. "Joseph": 11,
8. "Maria": 12,
9. "Gabriel": 10,
10. }

```

1. 这个字典中有多少个同学？可以参考下`len`函数。
2. 写一个函数，接受“ages”字典作为参数，返回的是其中年龄的平均值。遍历字典中所有的项使用的是`items`参数。
3. 写一个函数，接受“ages”字典作为参数，返回年龄最大的同学的名字。
4. 写一个函数，接受“ages”字典和数字“n”作为参数，返回的是一个新字典，新字典中每个同学的

年龄均是“n”。比如，`new_ages(ages, 10)` 返回的是“ages”字典的拷贝并且每个同学的年龄都是10。

子字典部分练习

先定义一个字典：

```
1. students = {  
2.   "Peter": {"age": 10, "address": "Lisbon"},  
3.   "Isabel": {"age": 11, "address": "Sesimbra"},  
4.   "Anna": {"age": 9, "address": "Lisbon"},  
5. }
```

1. “students”字典中有多少个同学？
2. 写一个函数接受“students”字典，返回平均年龄。
3. 写一个函数接受“students”字典和一个地址（address）参数，返回一个列表，内容是所有与传入的地址相匹配的同学的名字。

比如，调用 `find_students(students, 'Lisbon')` 会返回 `['Peter', 'Anna']`。

类

- 类
 - 类部分练习
 - 类继承
 - 继承部分练习

类

在面向对象编程（OOP）中，一个类就是一个允许一组属性和函数（一般称方法）的东西糅合在一起的一个数据结构。

下面定义的这个**Person**类定义了“name”和“age”属性，同时还有一个“greet”方法：

```
1. class Person:
2.     def __init__(self, name, age):
3.         self.name = name
4.         self.age = age
5.
6.     def greet(self):
7.         print("Hello, my name is %s!" % self.name)
```

大部分类都需要构造方法（`__init__`）来初始化类的一些属性。在之前的代码中，这个类的构造方法接受一个人的名字和年龄并将这些信息保存在类的实例中（使用 `self` 关键字访问的）。

“greet”方法则是打印存储在类实例（object）中的名字（name）。

类实例通过实例化到对象中使用。这里我们实例化两个对象：

```
1. >>> a = Person("Peter", 20)
2. >>> b = Person("Anna", 19)
3. >>> a.greet()
4. Hello, my name is Peter!
5. >>> b.greet()
6. Hello, my name is Anna!
7. >>> print(a.age) # We can also access the attributes of an object
8. 20
```

类部分练习

自行研究Python文档：<https://docs.python.org/3/tutorial/classes.html> 完成下列练习：

1. 写一个名为“Rectangle” 的类，存储给定长方形的坐标（x1, y1），（x2, y2）。

2. “Rectangle”类的构造方法接受4个参数（x1, y1, x2, y2）以**self**关键字存储在类实例中。
3. 给它写一个 `width()` 方法和 `height()` 方法，分别返回这个长方形的宽和高。创建两个“Rectangle”的实例对象来测试一下。
4. 再写一个 `area` 方法，作用是返回这个长方形的面积（宽*高）。
5. 最后写一个 `circumference` 方法，返回的是这个长方形的周长（宽*高*2）。
6. 写一个 `__str__` 方法，其返回内容是长方形坐标（x1, y1)(x2, y2)。然后打印所创建的类实例中的一个。

类继承

在面向对象编程中，继承是一种子类可以从另一个类中得到其属性和方法的形式，并允许其重写超类（被继承的类）的功能。

比如，基于我们上面写的“Person”类，我们创建一个子类，让这个子类的所产生的人年龄总为10：

```
1. class TenYearOldPerson(Person):
2.     def __init__(self, name):
3.         super().__init__(name, 10)
4.
5.     def greet(self):
6.         print("I don't talk to strangers!!")
```

在第一行中指出“TenYearOldPerson”类是“Person”类的子类。之后我们重写了子类的构造方法，让它只接受人的名字，但最终我们爱上要调用超类（被继承的类）的构造方法，参数是所传入的名字和10。最后我们重写了“greet”方法。

继承部分练习

使用之前定义的“Rectangle”类来进行下列练习：

1. 创建一个名为“Square”类，它是“Rectangle”的子类。
2. 为“Square”写上构造方法。构造方法应只含x1, y1两个坐标以及正方形的大小。记住，当你使用“super”来调用“Rectangle”的构造方法时你仍会用到这些参数。
3. 实例化两个“Square”对象，调用 `area` 方法并打印这个对象。确保所有的计算返回的是正确的结果，正方形的坐标始终与正方形的大小一致。

迭代器

- [迭代器](#)
 - [迭代器类](#)
 - [迭代器部分练习](#)

迭代器

正如我们之前看到的，在Python中我们使用“for”循环来迭代出对象中的内容：

```
1. >>> for value in [0, 1, 2, 3, 4, 5]:
2. ...     print(value)
3. ...
4. 0
5. 1
6. 4
7. 9
8. 16
9. 25
```

可以使用“for”循环（迭代）的对象称为迭代器。因此，一个迭代器也就是一个遵循了迭代协议的对象。

内置函数“iter”可以用来创建一个迭代对象，这时使用“next”函数可以来逐步迭代出内容：

```
1. >>> my_iter = iter([1, 2, 3])
2. >>> my_iter
3. <list_iterator object at 0x10ed41cc0>
4. >>> next(my_iter)
5. 1
6. >>> next(my_iter)
7. 2
8. >>> next(my_iter)
9. 3
10. >>> next(my_iter)
11. Traceback (most recent call last):
12. File "<stdin>", line 1, in <module>
13. StopIteration
```

当没有更多元素时，迭代器就会抛出“StopIteration”异常。

迭代器类

迭代器可以部署在类中。你只需要重写“`next`”和“`iter`”方法即可。

我们写一个模仿“`range`”函数的类作为例子，作用是返回所有“`a`”和“`b`”之间的值：

```
1. class MyRange:
2.     def __init__(self, a, b):
3.         self.a = a
4.         self.b = b
5.
6.     def __iter__(self):
7.         return self
8.
9.     def __next__(self):
10.        if self.a < self.b:
11.            value = self.a
12.            self.a += 1
13.            return value
14.        else:
15.            raise StopIteration
```

这样我们每次调用“`next`”时它都会让存储在内部的`a`前进，并且返回它的值。当它到达`b`时，就抛出“`StopIteration`”异常。

```
1. >>> myrange = MyRange(1, 4)
2. >>> next(myrange)
3. 1
4. >>> next(myrange)
5. 2
6. >>> next(myrange)
7. 3
8. >>> next(myrange)
9. Traceback (most recent call last):
10. File "<stdin>", line 1, in <module>
11. StopIteration
```

但最重要的是，你可以将迭代器类用在`for`循环中：

```
1. >>> for value in MyRange(1, 4):
2.     ... print(value)
3. ...
4. 1
5. 2
6. 3
```

迭代器部分练习

1. 写一个迭代器类，返回的是所有“a”到“b”之间的数的平方。
2. 写一个迭代器类，返回所有1到（n）之间的偶数。
3. 写一个迭代器类，返回所有1到（n）之间的奇数。
4. 写一个迭代器类，返回所有（n）到0之间的数。
5. 写一个迭代器类，返回的是从第一个元素到（n）之间所有的斐波那契数列。你可以回顾下函数部分了解下什么是斐波那契数列。

这是斐波那契数列的前几个数：`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...`。

6. 写一个迭代器类，返回的是所有0到（n）的连续对，如（0，1），（1，2），（2，3）...。

生成器

- [生成器](#)
 - [生成器部分练习](#)

生成器

如果你读了之前的章节，你就会知道带器是一个可以被“for”循环使用的对象。换句话说，迭代器就是遵循迭代协议的对象。

Python的生成器则是使用更简单的方法来部署迭代器的一个东西。不同于类，生成器是一个函数，每次遇到“yield”关键字时都会返回一个值。下面的例子是用生成器来生成两个数之间的数：

```
1. def myrange(a, b):
2.     while a < b:
3.         yield a
4.         a += 1
```

像迭代器一样，生成器可以用在“for”循环里：

```
1. >>> for value in myrange(1, 4):
2.     ... print(value)
3.     ...
4. 1
5. 2
6. 3
```

生成器与迭代器大同小异：

```
1. >>> seq = myrange(1,3)
2. >>> next(seq)
3. 1
4. >>> next(seq)
5. 2
6. >>> next(seq)
7. Traceback (most recent call last):
8.   File "<stdin>", line 1, in <module>
9. StopIteration
```

对于生成器来说，真正有点意思的地方是“yield”关键字。“yield”关键字很像“return”关键字，但不同于“return”的是，它允许函数恢复执行。换句话说，生成器所生成的每个值都是被需要的。Python见到“yield”关键字就会唤醒并恢复执行这个函数，如果函数还没有完全退出的话。生成器函数可以用在其他函数内，比如，配合“range”函数来生成一组数字再平常不过了：


```
1. def squares(n):  
2.     for value in range(n):  
3.         yield value * value
```

生成器部分练习

1. 写一个名为“squares”的生成器来生成 (a) 到 (b) 之间所有数的平方。使用“for”循环来测试。
2. 创建一个生成器来生成所有1到 (n) 之间的双数。
3. 创建另一个生成器来生成所有1到 (n) 之间的单数。
4. 创建一个生成器来生成所有 (n) 到0之间的数。
5. 创建一个生成器来生成斐波那契数列。范围是第一个数字到 (0)。斐波那契数列的前几个数字是： 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
6. 创建一个生成器来生成所有0到 (n) 之间所有的连续对。如 (0, 1), (1, 2), (2, 3)...

协程

- [协程](#)
 - [协程部分练习](#)
 - [管道](#)
 - [协程部分练习](#)

协程

Python的协程很像生成器，但并不是生成数据，大多数时候扮演了数据消费者的作用。换句话说，协程是一个在每次使用send方法发送数据后就会被唤醒的函数。

协程的技巧是将“yield”关键字写在表达式的右边。下面是一个打印出所发送的值的协程例子：

```

1. def coroutine():
2.     print('My coroutine')
3.     while True:
4.         val = yield
5.         print('Got', val)
6. >>> co = coroutine()
7. >>> next(co)
8. My coroutine
9. >>> co.send(1)
10. Got 1
11. >>> co.send(2)
12. Got 2
13. >>> co.send(3)
14. Got 3

```

首先要调用“next”来将协程推进。你可以看到它执行了一个打印。最终，函数进行到“yield”表达式了，然后就会等待唤醒。之后，每次有值被发送过来，协程就会从“yield”处唤醒，将值复制给val并打印出它。

使用 `close()` 方法可以关闭这个协程。

```

1. >>> co.close()
2. >>> co.send(4)
3. Traceback (most recent call last):
4.   File "<stdin>", line 1, in <module>
5. StopIteration

```

协程部分练习

1. 创建一个名为“square”的协程，它会打印出所发送的值的平方。
2. 创建一个名为“minimize”的协程，它会保存并打印出所发送的值中最小的一个值。

管道

协程可以用在部署数据管道中，其中一个协程会发送数据到下一个在数据管道中的协程。协程使用 `send()` 方法来将数据压入管道。



这是一个小型管道，值会从生产者协程发送到消费者协程打印出来：

```
1. def producer(consumer):
2.     print("Producer ready")
3.     while True:
4.         val = yield
5.         consumer.send(val * val)
6.
7.
8. def consumer():
9.     print("Consumer ready")
10.    while True:
11.        val = yield
12.        print('Consumer got', val)
```

正如上面所说，在发送任何数据前调用“next”是非常重要的一个步骤。

```
1. >>> cons = consumer()
2. >>> prod = producer(cons)
3. >>> next(prod)
4. Producer ready
5. >>> next(cons)
6. Consumer ready
7. >>> prod.send(1)
8. Consumer got 1
9. >>> prod.send(2)
10. Consumer got 4
11. >>> prod.send(3)
12. Consumer got 9
```

同样的，对于协程来说，数据可以被发送到不同的地方。下面的例子是部署了两个消费者，第一个它只会打印0-10之间的内容，第二个则是10-20。

```

1. def producer(consumers):
2.     print("Producer ready")
3.     try:
4.         while True:
5.             val = yield
6.             for consumer in consumers:
7.                 consumer.send(val * val)
8.     except GeneratorExit:
9.         for consumer in consumers:
10.            consumer.close()
11.
12.
13. def consumer(name, low, high):
14.     print("%s ready" % name)
15.     try:
16.         while True:
17.             val = yield
18.             if low < val < high:
19.                 print('%s got' % name, val)
20.     except GeneratorExit:
21.         print("%s closed" % name)

```

不要忘了调用下“next”呦。

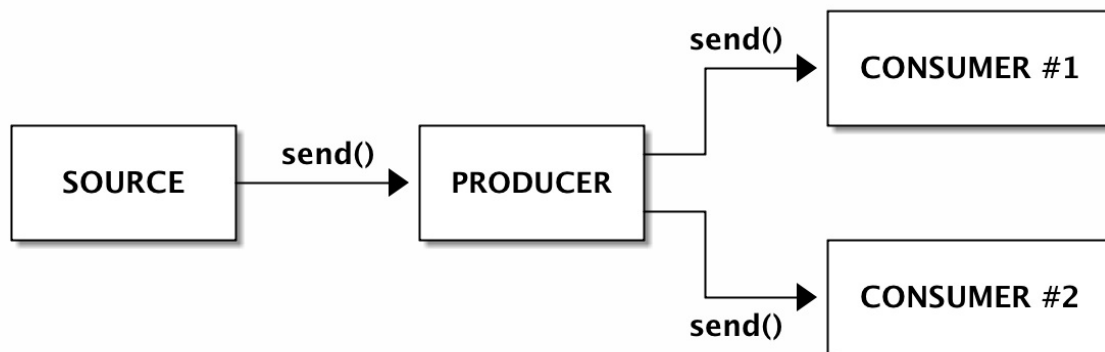
```

1. >>> con1 = consumer('Consumer 1', 00, 10)
2. >>> con2 = consumer('Consumer 2', 10, 20)
3. >>> prod = producer([con1, con2])
4. >>> next(prod)
5. Producer ready
6. >>> next(con1)
7. Consumer 1 ready
8. >>> next(con2)
9. Consumer 2 ready
10. >>> prod.send(1)
11. Consumer 1 got 1
12. >>> prod.send(2)
13. Consumer 1 got 4
14. >>> prod.send(3)
15. Consumer 1 got 9
16. >>> prod.send(4)
17. Consumer 2 got 16
18. >>> prod.close()

```

```
19. Consumer 1 closed
20. Consumer 2 closed
```

数据被发送到所有的消费者中，但只有第二个执行了打印命令。注意这里使用的“GeneratorExit”异常。一般都会用捕获异常的方式来通知下游协程这个管道没有用了，赶紧关闭了吧。



协程部分练习

1. 部署一个生产者-消费者管道，生产者会将值的平方发送给两个消费者。其中一个会储存并打印出所发送的值中最小的一个，另一个则是最大的一个。
2. 部署一个生产者-消费者管道，生产者会将值的平方发送给两个消费者，一次只发送给其中的一个。第一个值会发送给消费者1号，第二个则会发送给消费者2号，第三个又发送给消费者1号... 关闭生产者时会强制让消费者打印出它们所接收到的值的列表。

异步编程

- [异步编程](#)
- [异步编程部分练习](#)

异步编程

目前为止，我们在做的都是同步编程。同步编程执行过程很简单：一个程序从第一行开始，逐行执行一直到末尾。每次调用一个函数时，程序就会等待这个函数返回然后在执行下一行。

在异步编程中，函数地执行通常是非阻塞的。换句话说，每次你调用一个函数它就会立即返回，但相对得，这就表示函数并不会在其本来的位置被执行。它有了一种机制（名为 调度程序），让处在执行未来任务的函数随时可以由它进行响应。

异步编程的意思就是说在任何异步函数开始之前，程序都有可能结束。通常地解决方法是让异步函数返回“future（未来任务）”或者“promises（预先任务）”。让其标识出这是一个异步函数。最终，有调度程序的异步编程框架阻塞或者说等待这些异步函数完成它们的“future(未来任务)”。

自Python 3.6开始，“asyncio”模块与`async`和`await`关键字相结合，来让我们写多任务协作程序。在此类编程中，当一个协同函数在顽皮或者等待输入时，都会由`yield`交出控制权给另一个协同函数。

思考一下下面这个异步函数，它的作用是返回一个数字的平方，并且在返回前会睡眠一秒钟。

异步函数由 `async def` 声明。现在先忽略其中的 `await` 关键字：

```
1. import asyncio
2.
3. async def square(x):
4.     print('Square', x)
5.     await asyncio.sleep(1)
6.     print('End square', x)
7.     return x * x
8.
9. # 创建一个事件循环。
10. loop = asyncio.get_event_loop()
11.
12. # 执行异步函数并且等待其完成。
13. results = loop.run_until_complete(square(1))
14. print(results)
15.
16. # 将事件循环关闭。
17. loop.close()
```

事件循环 (<https://docs.python.org/3/library/asyncio-eventloop.html>) 是在有很多事物时, Python可以使用调度机制来执行这些异步函数。我们使用循环来让这些函数运行直到完成。

其中的打印语句是处于同步机制下的, 直到我们得到了任何结果 (`asyncio.sleep(1)`结束后) 剩下的才会执行。

之前的例子并不能很好地说明异步编程, 因为我们没有写得很复杂, 而且也只执行了一次函数。不过你可以想一下, 如果你要执行 `square(x)` 3次呢:

```
1. square(1)
2. square(2)
3. square(3)
```

因为 `square()` 里面有一个睡眠函数, 总执行时间差不多要3秒钟。但每次执行一个函数时计算机就会陷入呆滞, 在那一秒钟里什么也不做, 我们为什么不能让它在之前的那个函数睡眠时来执行下一个呢? 我们稍加改动, 把它变成异步:

```
1. # 执行异步函数直到其完成。
2. results = loop.run_until_complete(asyncio.gather(
3.     square(1),
4.     square(2),
5.     square(3)
6. ))
7. print(results)
```

一般来说, 我们使用 `asyncio.gather(*tasks)` 来让循环等待所有的任务完成。因为协同程序几乎会在同一时间内启动, 整个程序只需要1秒钟即可完成。要注意, `asyncio.gather()` 不会按顺序执行协同函数, 尽管它会返回一个按顺序排列的列表。

```
1. $ python3 python_async.py
2. Square 2
3. Square 1
4. Square 3
5. End square 2
6. End square 1
7. End square 3
8. [1, 4, 9]
```

有时我们需要立即处理所返回的结果。对于这种情况, 我们可以使用两个协同函数, 让它来处理结果, 使用 `asyncio.as_completed()` 就可以:

```
1. (...)
2.
3. async def when_done(tasks):
```

```
4.     for res in asyncio.as_completed(tasks):
5.         print('Result:', await res)
6.
7. loop = asyncio.get_event_loop()
8. loop.run_until_complete(when_done([
9.     square(1),
10.    square(2),
11.    square(3)
12. ]))
```

打印出的东西差不多是这样的：

```
1. Square 2
2. Square 3
3. Square 1
4. End square 3
5. Result: 9
6. End square 1
7. Result: 1
8. End square 2
9. Result: 4
```

最后要说的是，一个异步函数可以使用 `await` 关键字来调用另一个异步函数。

```
1. async def compute_square(x):
2.     await asyncio.sleep(1)
3.     return x * x
4.
5. async def square(x):
6.     print('Square', x)
7.     res = await compute_square(x)
8.     print('End square', x)
9.     return res
```

异步编程部分练习

1. 写一个异步协同函数，它会将两个变量相加，并且会睡眠这些时间。使用异步循环来调用它。
2. 修改一下之前的那个程序，让它调度两次。