简单粗暴 Python 3

Kanglong Wu
<wklchris@hotmail.com>

July 22, 2016

Contents

Chapter 1

序言：还是需要看一下的

这一章的部分内容你可以到 Python 的官网 FAQ 上面寻找答案。

General FAQ: https://docs.python.org/3/faq/general.html.

1.1 Python 的特性

Python (www.python.org) 是一门解释性编程语言，而我们熟知的入门编程语言可能是以编译性为主（比如 C 语言）。最显著的特点之一就是 Python 学习起来更简单，或者说异常简单。因为解释性的语言往往可读性更好。此外的一个特点就是，你不能用 Python 直接生成一个可执行的软件（像 exe 扩展名文件之类），所以它往往以**脚本（Script）** 的形式存在，并一般只在安装了解释器（也就是 Python）的计算机上运行。

多的赞美 Python 的话就不必说了，随着深入你会明白的。

1.2 库、包和模块

库 (package)、**包 (Library)** 和**模块 (module)** 是 Python 中重要的概念，也是其健壮的根源。模块的后缀名都是.py，而包呢，一般就是由一些模块文件组合成的一个文件夹，并在文件夹内放一个空的 `__init__.py` 文件。库则由许多包组成。Python 的 Main 文件后缀名也是.py。至于怎么调用模块我们在后续章节提到。

Python 的库和模块多数能在：pypi.python.org/pypi这里，通过右上角的搜索来找到。你可以尝试搜索 NumPy, SciPy.

1.3 Python 版本与发行版

Python 一般有 2.x 与 3.x 两大分支版本号。也不一定说 2.x 版本的 Python 就一定在性能上劣于 3.x，不过 2.x 的最终大版本定格在 Python 2.7，因此说 Python 3.x 才是 Python 的未来丝毫不过分。

发行版（distribution）是指 Python 相关的功能的版本，比如科学计算发行版 Anaconda。

在 SciPy 的官网的这个页面，你可以找到一些用于科学计算的环境推荐：比如 Anaconda, Enthought, Python(x,y) 都是非常优秀的环境。这些内容的安装与否，可以在阅读了科学计算的教程后再做决定。

1.4 Python 编辑器和集成开发环境

Python 的编辑器与其他语言没有什么不同，都只是打开.py 文件编辑罢了。就算拿记事本也能当 Python 编辑器用。这方面著名的有大名鼎鼎的“撕逼双雄”Vim 和 Emacs, 以及 Sublime, Eclipse, NotePad++ 等等。编辑器一般支持语法高亮就算能用了。

随着 Python 安装到你的电脑，你能在开始菜单发现一个叫 IDLE 的东西，那个就姑且算 Python 自带的编辑器，不过是交互式的。也就是说，命令基本上只能一句一句输入，但是每一句输入都能看到结果，之前输入的语句也会被记录。虽然 IDLE 很简陋，但是在学习 Python 的过程中，你需要经常用到。在本手册中，IDLE 里面的命令会以三个大于号（>>>）开头。实际上在 IDLE 中你看到的也是这样。

编辑器一般不好配置运行程序，因为你要告诉编辑器 Python 被安装在哪里、去调用 python.exe 来运行你编写的.py 程序。因此**集成开发环境（Integrated Development Environment, IDE）**能够既提供编辑器的功能，又为你做好这些配置工作，甚至实现让你推送更新到 GitHub、调用 PyQt 的 Designer 等等功能。可以说一个集成开发环境就让你高枕无忧。IDE 当之无愧的业界一哥的是 Visual Studio，但就 Python 这一个语言来说，PyCharm 是最受推崇的。

1.5 本手册的背景

笔者使用的是 Python3.4，其他安装的库会在后续提及。操作系统是 Windows。

笔者的 IDE 是 PyCharm(下载)。虽然体积稍微有点大（当然比 Visual-Studio 不知道小到哪里去了），但用过都说好。

1.6 学习资料推荐

廖雪峰先生的教程一直以通俗易懂闻名，廖雪峰 Python 3 教程，当然里面也有之前的 Python 2 的教程。廖先生 Python 3 教程的 GitHub 源码部分在这里。

其他的教程见仁见智吧，不同的模块有不同的教程。

比如数据处理方面，你也许听说过 O'Reily 的 *Python For Data Analysis*(PFDA)，中文译版叫《利用 Python 进行数据分析》。可惜这本书是 2013 年的，比较老了，介绍的是 Python 2. 不过前面我也说过，Python 版本间的区别是可以克服的。

最近我发现了 Amit Saha 著的 *Doing Math with Python* 这本书，写于 2015 年。它使用的版本是 Python 3，这在现在市面的 Python 数据书是非常难得的。不过它讲解没有 PFDA 一书那么深入，而且“数学”和“数据”还是有区别的——但大体值得一读。

Chapter 2

基础

2.1 面向对象，万物皆类

面向对象编程最重要的概念就是：**类（class）**。在 Python 里面更为极端：**Python 里的所有东西都是类**。

比如 a = 1，你会发现 a 也是一个类。你可以用 `dir()` 去处理它，可以看到它的成员，包括**属性（Attribute）**和**方法（method）**。

2.2 函数调用

与大多数编程语言一样，最基本的函数调用方式是：

$$< 关键词 >(< 参数\ 1>, < 参数\ 2>, ...)$$

例如：

```
1 print('Hello world')
```

Python 每行的语句结尾不需要有分号之类的东西。简洁至上。

2.3 语段环境：缩进

Python 不借助于像花括号“{ }”或者 end 关键词一类的标志来标定一个语段的结束。它使用缩进控制语段。比如一个 if 语段：

```
if 2 > 1:
    print('Wow')
```

所以在 Python 中，**行首的空格不能随意书写**。每一个下层的语段，应当相对其父语句向右缩进 4 个英文字符。可以看出 Python 对于代码撰写格式的执着，也体现了 Python 认为代码的可读性是非常重要的事情：**代码是给人看的，只是偶尔需要运行一下。**

注意，父语句的句末有一个冒号。

2.4 简化的变量

在 Python 中，你不需要提前声明变量，也不需要对变量指定类型。变量一经赋值直接可以使用。其原因是在 Python 中，变量只是指向内存数据的一个引用。

```
a = 'Hello'
print('a')
a = 1    # 直接赋值为另一数据类型
print(str(a))
```

注意，Python 变量复制中的一个问题，就是**某些类型（不包括简单的如数字和字符串类型）的数据复制以后，其实只是复制了一个“引用”，而不是复制了一个“样本”**。这导致原有的变量更改会使得之前复制的变量也发生变动。你可以在 IDLE 中尝试这个：

```
>>> a = [1,2]  # 这是一种叫列表list的数据结构
>>> b = a
>>> id(a)       # 变量a对应数据的内存地址
944222300616
>>> id(b)       # 两者地址是一样的，一个改动另一个也会改动
944222300616
>>> b.append(3) # 把b后面添加一个数字
>>> a
[1, 2, 3]
>>> a is b      # 判断变量a、b指向的数据是否同一个
```

```
True
```

如果你想弃用变量 a，使用 `del(a)`. 注意，只要一个内存数据仍然有变量指向它，它就不会被抹除。除非你删除所有指向该数据的变量。

2.5 注释和续行

在 Python 中，注释就是用一个 '#' 开始的。如果是多行注释，用三个单引号包括即可。至于三个双引号，在方法定义中有特殊的用途——所以单纯的注释，请使用单引号。

```
# 这是一个行内注释
'''
这是行间注释
'''
```

在编辑器或者 IDE 中，一条语句太长了，想换行写？在行尾输入一个反斜杠和一个空格：'\ '，然后下一行可以继续书写。

有时候，在行尾有明显的未完成语法，也能在下一行直接续写。比如：

```
a = ['This string is extremely looooooooooooong',
     'And this also']
```

2.6 打印与输入

打印就是上文使用过的 print 函数，你可以一次打印多个数据，Python 会自动在它们之间插入空格：

```
>>> print(2, 3)
2 3
```

input 则是从键盘向 Python 中的字符串赋值的函数，其参数是一个字符串，在请求键盘输入时会显示在屏幕上：

```
>>> a = input('How old are you?')
How old are you?Shut up!
>>> a
'Shut up!'
```

2.7 编码

编码是个复杂的问题，一般推荐使用 utf-8 编码，并在.py 文件的第一行书写：

```
# coding: utf-8
```

你可以通过 help(str.encode) 来查看更多内容。本手册不作详细介绍。

2.8 基于二进制的浮点运算

这一部分主要是要告诉你，Python 并不是一拿来就能做数据的！

```
>>> 0.1 + 0.2
0.30000000000000004
>>> round(1.2345,3)      # 四舍五入到小数点后第3位
1.234
```

等等！这好像哪里不对！我书读的少，连 Python 也骗我?!

其实如果不信的话，你大可以尝试一下在 IDLE 中输入这两个语句。这是由 Python 的运算机理决定的。其他同机理的编程语言也有一样的问题。

那有没有解决方法呢？有的。比如 python 自带的 decimal 模块和 fraction 模块，以及第三方的 NumPy 库，都是能够解决运算问题的。我想说的是：请不要在没准备好的情况下用 Python 做运算。仅此而已。Python 也不是生来就是做计算的，有模块的情况下能解决计算问题，就不用过于苛责于未加载模块时的这些问题了。

2.9 其他

最后，你想查看某个变量或者类的下属成员用法，请活用 dir() 和 help() 命令。甚至你有相当一部分学习储备会来自这里。当然，你也可以到官网的这个页面来学习。

Chapter 3

数据结构

这里的数据结构与 Python 中的并非一节对一个的对应关系。比如浮点和整数都放在了数字这一节。具体请用 type(<variable>) 命令查看：

```
>>> a = 1
>>> type(a)  # class？还记得“万物皆类”吗？
<class 'int'>
```

3.1 数字

分为两种：1) 整数，int；2) 浮点数，float。先说四则运算：+，-，*，/，乘方是 **. 四舍五入到小数点后第 n 位用 round(<num>, n) .

然后整数和浮点数运算有这些规律：

1 - 运算的两数中只要有一个浮点数，结果就是浮点数。

2 - 整数相除，即使能整除，结果也是浮点数。

3 - 取商：5//2 → 2，取余：5%2 → 1，也可以一起做：divmod(5,2) → (2,1).

4 - 你可以使用自增方法： a += 1 ，且-=，*=，/=，**= 均可。

5 - Python 的机制使其可以自动解决整数溢出问题，这方面毋须操心。

更多的运算在 math 模块中有提及，在文件开始使用 import math 来加载此模块。你可以在 IDLE 中，使用 dir(math) 来查询该模块定义的函数，并用 help(math.pow) 这样的语句来调出该函数的用法说明。

3.2 布尔和逻辑

布尔型（bool）有两个值：True 和 False，注意首字母大写，其余小写。逻辑运算符与或非：A and B, A or B，not A. 等于和不等于分别是 `==`，`!=`。

可能会经常用的的逻辑判断：`bool(a)`，或者在 if 语句中直接 `if a:`，判断变量 a 指向的数据是否为空。注意这个比 `a is None` 更加严格，因为在 `a=[]` 时前者是 False 而后者是 True.

3.3 序列

Python 中的**序列（sequence）**主要包括**字符串 (str)**、**列表 (list)** 和**元组 (tuple)**。序列是指一串排列有序的数据，能够通过索引（也就是元素在序列中的序号）来取出其中的数据。在同一时刻，序列的长度若为 N，索引范围就是小于 N 的所有自然数（包括 0），因此序列的第一个元素总是: seq[0]。Python 中，允许使用负数索引序列，比如 seq[-1] 表示序列最后一个元素。

你还可以使用 seq[1:3] 或者 seq[-3:-1] 的方式来获取 seq 的一个“切片”，其起止是 seq[1] 到 seq[2]，不包括 seq[3]. 你也可以使用 seq[1:], seq[:5], seq[:] 这些方式，相信你能够理解它们的含义[1]。不过一般很少使用切片来参与数据传递。因为在 Python 2 中，列表的切片结果与原列表同址，故更改原列表会引起新列表变动。在 Python 3 中它们已不再同址。

序列有一些共用的操作方法 (更多的，可以参考zip和enumerate)：

len() 返回序列长度，比如字符串长、列表元素个数。

+/* 用加号连接两个序列，乘号能够重复排列若干个序列然后连接它们。

in 返回 `seq1 in seq2` 的查找结果，若 seq2 中包含 seq1 这个片段，就返回 True，否则返回 False.

index 只有在 seq1 in seq2 = True 时才能使用。seq2.index(seq1) 表示 seq1 **首次**出现在 seq2 中的位置，即 seq1[0]=seq2[seq2.index(seq1)].

max()/min() 返回序列中 ASCII 码的最值对应的数据元素。

[1]不过请注意，`seq[:5]` 这种方式给出的是 `seq[0]` 到 `seq[4]`，而不是到 `seq[5]`.

cmp 返回 cmp(seq1,seq2) 两个序列的比较结果。小于 0 表示 seq1 较小（ASCII 码），等于 0 表示相等，大于 0 表示 seq2 较小。

3.3.1 元组

元组是一种不可变更的序列，用圆括号来创建。

```
>>> a = (1, 'string', [1,2])     # 可包括多种数据类型
>>> a[0]
1
>>> b = ('a',)   # 单元素的元组
```

3.4 序列：列表

列表（list）属于序列。由于重要性，作同级章节撰写。创建比如：

```
>>> lst = [1, 2, 3]
```

除了在序列章节提到的序列共有方法，还有：

反转 使用 `lst2 = list(reversed(lst))` 来产生一个顺序相反的列表。如果你想更改原有的列表而不是产生新列表，使用 `lst.reverse()` .

追加 使用 `lst.append(4)` 来追加一个元素，使用 `lst.extend([5,6])` 来追加另一个列表。

插入 使用 `lst.insert(1, 'd')` 来在 `lst[1]` 处放置新的元素'd'，原有的 `lst[1]` 后移成为 `lst[2]` ，其后依次后移。

删除 使用 `lst.remove('d')` 会搜索 lst 中是否含元素'd'，搜索到的**首个匹配结果**会被删除；如果没有搜索结果就报错。而 `lst.pop(2)` 会**返回** `lst[2]` ，然后将其从 lst 中删除。如果括号内省略 2，它将自动设置为-1，即返回最后一个元素，并将其从 lst 中删除。如果指定的元素不存在，报错。

搜索 相比于所有序列都能使用的 `index()` 方法，使用 `lst.count('a')` 可以返回数据在 lst 中出现的次数。在不含该数据时，`count()` 会返回 0，而 `index()` 则会报错。

排序 使用 `lst.sort()` 就能按升序排列列表，而 `lst.sort(reverse=True)` 则可以按降序排列。

清空 使用 `lst.clear()` .

列表有个很强大的功能叫”列表解析“，参见for 语句这一节。

3.5 序列：字符串

字符串属于序列。虽然字符串应该属于序列的子章节，但是由于比较重要，因此作同级章节撰写。

字符串可以包括在一对单引号内，或者一对双引号内。这两种方式 Python 都是承认的。比如：

```
a = "Hello world"
b = 'Hello world'
c = "What's your name?"
d = 'What\'s your name?'          # 使用了转义字符
e = a + b
f = c * 2         # 两个字符串c拼接成f
g = str(12345) # 强制格式转换
a += '!'          # 自增一个字符
```

3.5.1 转义字符和直接输出

上例中提到了转义字符，之前的续行也提到过‘\ ’，实际上也是转义字符。Python 的常用转义字符列表如下：

\␣ - （行末）续行	\\ - 反斜	\a - 响铃	\b - 退格	\000 - 空
\n - 换行	\v - 纵制表	\t - 横制表	\r - 回车	\f - 换页

转义字符有时会以你不希望的方式执行，例如输出一个文件夹路径：

```
>>> print('c:\new')
c:
ew
```

这里 Python 把\n 视作了转义字符，然后换行了。实际上，我们有前加字母 r 的方式来让 Python 不对字符串内的内容做任何解释，直接输出。

```
>>> print(r'c:\new')
c:\new
```

3.5.2 字符串常用方法

除了在序列这一节中提到的方法，字符串还常常用到：

```
>>> s = ' I love Python'          # 首个字符是空格
>>> lst = s.split(' ')  # 用空格切割字符串s为一个列表
>>> lst
[ ‘’ , 'I', 'love', 'Python']
>>> ','.join(lst)        # 用逗号连接lst中的元素为一个字符串
'I,love,Python'
>>> s.strip()   # 去掉字符串s两侧的空格
'I love Python'
>>> s.title()   # 每个单词首字母大写
'I Love Python'
>>> s
' I love Python'
```

注意：以上方法都不会改动原来的字符串，只是返回了一个新的字符串。原来的字符串是不会受到影响的。

此外，与 strip() 类似的 lstrip()/rstrip() ，可以分别去掉字符串左侧/右侧的空格。而方法 upper()/lower()/capitalize() ，分别能够使字符串转换为全大写/全小写/句首大写。

3.5.3 字符串格式化

字符串格式化是很有用的一个功能，先举例说明：

```
>>> 'I like {} and {}'.format('Python', 'you')  # 不编号
'I like Python and you'
>>> '{0} + {2} = {1}'.format(10, 10, 'Python')  # 灵活引用
'10 + Pyhton = 10'
>>> '{0}{1}{0}'.format('abc',2) # 编号可以反复调用
```

还有一些复杂的格式化技巧，例如：

```
>>> '{:0>4d}'.forma(1)  # 输出四位数，左侧不足则补0
'0001'
>>> '{:b} and {:1e}'.format(5,100) # 转二进制和科学计数
'101 and 1.0e+02'
```

下面一个表格教你如何像上面一样进行格式化：

:s	-	字符串
:c	-	单个字符
:b, :o, :x, :d	-	二、八、十六、十进制数
:e, :f	-	科学计数法；浮点数
:.2f，:+.2f	-	两位小数，带符号两位小数
: .2f	-	正数前补空格的两位小数
:,	-	使用逗号分隔符
:.2%	-	带百分比的两位小数
:.2e	-	科学计数法下两位小数
:>8d/<8d/^8d	-	总宽 8 位居左/居右/居中对齐，其中 > 号可省略 在 >/</^ 号左边可以输入单个字符使其成为补位符

还有从数据读入的方法，比如从序列、字典，甚至从类中读成员属性：

```
>>> s = [1, 2, 3]
>>> '{}, {}, and {}'.format(*s) # 序列：单星号
'1, 2, and 3'
>>> s = 'abc'
>>> '{0}, {2}, and {1}'.format(*s)
'1, 3, and 2'
>>> d = {'name': 'wkl', 'gender': 'male'} # 字典：双星号
>>> 'Gender of {name} is {gender}.'.format(**d)
'Gender of wkl is male.'
>>> c = 3 - 5j  # 复数3-5i，用dir(c)可知它有real/imag成员
>>> 'Real: {0.real}, Imaginary: {0.imag}'.format(c)
'Real: 3.0, Imaginary: -5.0'
```

3.5.4 正则表达式

Python 中专门有个模块负责正则表达式。为了完整地介绍这一内容，请阅读re 模块这部分的内容。

3.6 字典

上文中提到了字典的概念。字典的几种创建方式：

```
>>> d1 = {'name': 'wkl', 'gender': 'male'} # 定义
>>> d2 = {}
>>> d2['name'] = 'wkl'  # d['key'] = value
>>> key_value = (['name', 'wkl'], ['gender', 'male'])
>>> d3 = dict(key_value)          # 从元组创建
>>> d4 = {}.fromkeys(('name', 'gender'), 2333) # 多对一
>>> d4
{'name': 2333, 'gender': 2333}
```

均符合**键值对（key-value pair）**的创建原理。上述的 'name' 和 'gender' 叫做**键（key）**，对应的 'wkl' 和 'male' 分别称为这两个键的**值（value）**。键是字符串，而值可以是任意的数据类型。字典数据都以键值对的形式存在，即每个键只对应一个值。字典无序，因此它没有类似 d[0] 的索引。

字典的操作方法：

```
>>> d = {}.fromkeys(('name', 'gender'), 'N/A')
>>> dd = d.copy()          # 复制字典d到新字典dd
>>> dd is d     # False说明更改dd和d其中的一个不会影响另一个
False
>>> d
{'name': 'N/A', 'gender': 'N/A'}
>>> len(d)       # 字典长度
2
>>> 'name' in d # 查找字典中是否有key='name'
True
>>> del(d['name']) # 删除key='name'的键值对
>>> dd  # 无影响。可以试验dd=d.copy()与dd=d的区别
{'name': 'N/A', 'gender': 'N/A'}
>>> d.get('name')  # Key='name'已删除，无返回值
>>> d.get('name', 'Nothing')     # 无对应key则返回'Nothing'
'Nothing'
>>> d.setdefault('name') # 返回key='name'对应的value
>>> d   # 注意此时已经新增了key='name'，不过value为空
{'gender': 'N/A', 'name': None}
>>> del(d['name'])
>>> d.setdefault('name', 'wkl') # 无对应key，则新增该key
>>> d
{'gender': 'N/A', 'name': wkl}
```

```
>>> list(d.items()) # 以元组(key, value)组成list输出字典
[('name', 'wkl'), ('gender', 'N/A')]
>>> list(d.keys()) # 以字符串组成list输出字典的所有键
['name', 'gender']
>>> list(d.values()) # 对应d.keys()中的顺序输出所有值
['wkl', 'N/A']
>>> d.pop('name') # 返回'name'对应的值，并删除该键[2]
'wkl'
>>> d.update(dd) # 以字典dd为准，更新字典d中的key=value
>>> d
{'name': 'N/A', 'gender': 'N/A'}
```

关于字符串如何用字典进行格式化，参考这里。

3.7 集合

集合是一种无序的数据存储方式，且内部元素具有唯一性。集合与字典一样都可以用花括号的形式创立。但在书写 `a={}` 时，Python 会将其识别为字典类型。

集合的创建方法[3]：

```
>>> s = set('abc') # 从序列创建，每个序列元素作为集合元素
>>> s  # 你当然也能像下面输出的这行一样创建集合
{'a', 'c', 'b'}
```

集合的一些操作：

增添 使用 add 或者 update 方法。

删除 使用 remove 或者 discard 方法。区别在于后者找不到元素会报错。

从属 判断相等： `a == b` .
判断 a 是否为 b 的子集/父集 `a.issubset(b)/a.issuperset(b)`.

运算 并：$a|b$ 或者 a.union(b)，交：$a\&b$ 或者 a.intersection(b)，补：$a-b$ 或者 a.difference(b). 注意以上 6 个方法都不会改变集合 a 或 b 本身。

[2]如果使用 d.popitem(<key>) 方法，其效果与 pop(<key>) 相似。但是 popitem() 可以不带参数，效果是从字典中随机选择，将其返回后再将其删除。

[3]本手册只讨论可变集合。不可变集合请 `help(frozenset)` .

Chapter 4

语句、迭代器与生成器

语句的概念就不用多说了，if/for/while 三大元帅，还有一些不止是虾兵蟹将的阵容，也很值得一看。迭代器和生成器是 Python 里面很特殊的内容，你平时不一定会使用，但是了解是非常必要的。

4.1 if 语句

一个例子结束。其中 elif 相当于很多编程语言里的 else if . 如果没有写 else 这个语段的必要，那就省略。

```
if 1.0 > 1:
    a = 1
elif 1.0 < 1:
    a = 2
else:
    a = 3
```

形如 a = X if flag else Y 的操作叫做三元操作：当 flag 为真，把 X 赋给 a；否则把 Y 赋给 a. 例如：

```
>>> a = 'this' if 2 < 1 else 'that'
>>> a
'that'
```

4.2 for 语句

语法不用多说，直接例子。比如 for 语句用于序列：

```
>>> a = 'hi'
>>> for a_index in a:
...     print(a_index)  # 行首四个空格
...
h
i
```

再比如字典：

```
>>> d = {'name': 'wkl', 'gender': 'male'}
>>> for k, v in d.items():  # 也可以用for k in d.keys():
...     print(k, v)         #             print(k, d[k])
...
gender male
name wkl
```

4.2.1 range 函数

介绍一个函数 range(M, N, s)，它也经常会被用作 for 语句的助手中。有三种调用方式：

- range(N)：等同于 range(0, N, 1)
- range(M, N)：等同于 range(M, N, 1)
- range(M, N, s)：返回一个介于数 M 和数 N 之间、步长为 s 的整数序列。注意：如果 $N > M$，那么应有 $s > 0$. 如果 $M > N$，那么应有 $s < 0$. 以上情形下，返回的序列都不包含数 N 在内。比如 list(range(3, 1, -1)) → [3,2].

一般而言，range() 输出的结果尚且不是一个序列。但你可以用 list(range()) 这样将其转为序列。

4.2.2 列表解析

这个功能实在是太方便了，以至于你知道怎么写 for 语句就应该知道怎么用它创建列表。

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

甚至这个功能可以用在文件读取上。参考这一小节。

4.3 while 语句

在 Python 里， while 语句支持一个附加的 else 语句块，意思是退出循环后就执行那部分的内容。当然，不需要你可以不写这个语段。

```
count = 1
while count < 5:
    a = count
    count *= 2
else:
    b = count
```

结果是 $a = 4, b = 8$. 如你所见，在 else 语句中，循环自增器的值被保留了。

4.3.1 zip 函数

zip 函数的参数必须是多个序列。它能够返回参数中按次序组合而成的一个列表，该列表的基本元素是元组，且其长度取决于参数中最短的序列。

如果说人话，就是这样：

```
a = [1, 2, 3, 4]
b = [5, 6, 7]
c = 'abcde'
lst = list(zip(a,b,c))
print(lst)
```

结果是： [(1, 5, 'a'), (2, 6, 'b'), (3, 7, 'c')] .

看起来好像很厉害的样子……它有什么用呢？比如你想把列表 a 和 b 对应元素相加：

```
a, b = [1, 5, 'a'], [2, 6, 'b']
result = []
for x, y in zip(a, b):
    result.append(x + y)
print(result)
```

结果是：`['3', '11', 'ab']`.

你也许觉得上面那个例子太不值一提了，那么如果你想交换字典中 key 和 value 的位置：

```
d = {'name': 'wkl', 'gender': 'male'}
print(dict(zip(d.values(), d.keys())))
```

结果：{ 'wkl': 'name', 'male': 'gender' }.

4.3.2 enumerate 函数

返回序列的由每一个索引和对应数据组成的元组构成的列表。

```
for index, data in enumerate(lst):
    print('lst[{0}] = {1}'.format(str(index), str(data)))
```

你也可以使用如 `enumerate(lst, 1)`，指定它从 lst[1] 开始向后索引。

4.4 迭代器

如果我们想读一个列表，可以这样：

```
s = list('abcdef')
while s: # 如果s非空，输出它的首元素
    s.pop(0)
```

迭代器（iterator）则是这样:

```
s = list('abcdef')
s = iter(s) # 将序列s变成一个迭代器
while True:
    s.__next__()
```

当然，上面这段代码读到最后会报错（StopIteration），但是 s 中的值还是可以正常输出的。报错的原因在于其内部的指针一直在移动，到了迭代器的尾部后，再往后移动指针就无法正常返回数据了。如果你在面向对象编程中学习过链表这种数据结构，对于里面节点 Node 有印象的话，你就知道 Node 这个类在定义的时候有个叫 next() 的方法[1]，指向的也是一个 Node. 很明显链表就是一个迭代器。

迭代器做的就是按顺序一个个输出数据。类似地理解，`for` 语句可以被称为迭代工具。实际上，`for` 语句的工作过程中，正是用 `__next()__` 来处理的。**迭代器的优势在于并不是一次读入所有数据，而是分步读入的。这对于内存优化很有意义。**

还记得 `range()` 函数吗？这里有个迭代形式的 `xrange()` 函数。比如执行 `zip(range(4),xrange(100000000))` 这个代码，`xrange()` 实质上只生成了 4 个值而不是一亿个值。这就是迭代器的内存优势。

最后，怎么判断一个内容是否是迭代器？用 `dir()` 方法。如果结果中有 `__iter__` 这个成员，那么它就是可迭代的。

4.5 生成器

来看一个**生成器（generator）**的简单创建方式：

```
generator = (x**2 for x in range(1,4))
```

哦对了，为了防止你把它认成列表解析，我得说一句：列表解析最外侧用的是方括号，生成器则用的是圆括号。

但是这跟列表解析相比又有什么优势呢？其实生成器也完全具有迭代器的特征，因此优势在于读取大文件或者处理大数据时的内存优化。

想要真正地理解生成器，你可以先阅读函数定义部分的内容。不过没关系，告诉你 `def` 语句是用来申明一个函数的，那么一个生成器就是指：**在定义中含有 yield 语句的函数**。当然了，上面的那个生成器例子含有一个隐式的 yield 函数。

```
def func(n):
    while True:
        yield n*2
a = func('w')
```

[1]多说一句，在 Python 2 中，调用方式其实就是 `next()` 而不是 `__next()__`.

```
print(a.__next__())
b = a.send('abcd')  # 执行a.send前a必须已经执行过__next()__
print(b)
a.close() # 关闭生成器
```

你可能会说，这个 yield 与 return 很像嘛。的确， yield 的作用就是返回一个相应的值，然后将生成器挂起，等待下一个指令。生成器的工作机理：

1 - 若没有被从 __next__() 方法调用，生成器不启动。

2 - 检测到 __next__() 方法调用，生成器启用，并逐句执行，直到 yield 语句。挂起。

3 - 从 yield 语句之后继续执行，直到下一个 yield 语句。如果下一个不存在，那么返回 StopIteration 错误。

4 - 保持挂起状态，直到有 close() 方法关闭此生成器。

有什么应用的例子吗？就拿斐波那契数列为例吧：

```
def fib(N):
    n, former, later = 0, 0, 1
    while n < N:
        yield later
        former, later = later, former + later
        n += 1

numlst = []
for each_fib in fib(10):
    numlst.append(each_fib)
print(numlst)
```

输出： [1, 1, 2, 3, 5, 8, 13, 21, 34, 55] . 如上，生成器即使不用 __next__() 方法，也是有其独到之处的。

Chapter 5

错误

错误在编程中经常出现。Python 的常见错误主要包括以下几类：

ZeroDivisionError 除数为 0

SyntaxError 语法错误

IndexError 索引超出序列范围

KeyError 字典键不存在

IOError 读写错误

在面对错误风险时，我们常常需要 try 语句。格式：

```
try:
    <尝试执行...>
except (Error1, Error2) as e:
    print(e)
    exit()
else:
    <如果没有发生错误，执行...>
finally:
    <无论是否有错误，均执行...>
```

其中， except 的错误类型如果不能确定，可以不写。如果不想输出错误信息，可以去掉 as e 以及相关语句。

最后， else 和 finally 语段都不是必要的。

Chapter 6

文件

6.1 open 函数

Python 中操作文件的方法是 `open(file, mode='r', encoding=None)`. 其实还有很多很多的参数，但是一般也用不到了。本手册对于 encoding 也不太多做介绍，这个和之前介绍编码方式的 decode 属于同一部分的内容，请读者自行查阅资料。

其中 `file` 参数没什么好说的，比如`r'c:\new\f.txt'` 之类的。如果路径已经更改到文件所在的路径，你也可以省略路径直接写`'f.txt'` 即可。

至于 `mode='r'` 参数，这里也说明了默认是'r'. 各参数的含义如下：

'r'	-	读（默认）。
'w'	-	写。该模式会删除原有内容。如果不存在文件会创建一个。
'x'	-	创建新文件并写。
'a'	-	追加。
'b'	-	二进制模式。
't'	-	文本模式（默认）。
'+'	-	打开文件以读写。

6.2 读写和其他方法

先看怎么操作一个文件。一种是从 Python 2 一直延续的：

```
f = open(r'e:\f.txt')
for line in f:
    print(line) # 输出每一行
f.close() # 这一行不能忘
```

后来大家说这个方法不好。为啥？ close() 容易忘啊。所以 Python 3 里面提供了一种不需要写 close() 的方法，也是更推荐的方法：

```
with open(r'e:\f.txt') as f:
    for line in f:
        print(line)
```

上面的两种方法都是针对一行一行读文件而言的。实际上文件不一定需要循环语句来帮忙读取。比如：

```
with open(r'e:\f.txt') as f: # 假设文件两行分别是'1'和'2'
    a = f.read() # 结果是'1\n2'
    b = f.readlines() # 结果是['1\n', '2']
```

除了 read()/readlines() 这两个一次性读取整个文件的方法，还有一个叫 readline() 的方法，利用迭代器的原理，一行一行读取：

```
with open(r'e:\f.txt') as f:
    a = f.readline() # a='1\n'
    b = f.readline() # b='2'
```

眼熟吗？没错， readline() 每次执行的结果不同，这说明是 f 一个迭代器啊。实际上，你早就知道通过 open 方式读取的文件是迭代器了——因为之前介绍过 for line in f 这样的语句。

迭代器的一大麻烦在于指针的位置。迭代一遍后到了文件尾，那我怎么再读一遍这个文件呢？这里就要用到 seek()/tell() 这两个方法。

```
with open(r'e:\f.txt') as f:
    f.readline()
    print('现在读到第{}个字符'.format(f.tell()))
    f.seek(0) # 现在指针回到了第0个字符后
    f.readline()
```

然后写入，这个就没那么多讲究了：

```
with open(r'e:\f.txt', 'w') as f:
    f.write('Sometimes naive.')
```

6.2.1 查看文件属性

通过 os 模块的 stat 方法来查看文件的创建日期、修改日期等属性。这一块不过多介绍，请自行查阅文件。

```
import os
os.stat(r'e:\f.txt') # 查看属性不需要open文件
```

6.2.2 大文件的读取

大文件用 read()/readlines() 方法会增加内存的负担。所以使用 readline() 方法：

```
with open r'e:\f.txt' as f:
    while True:
        line = f.readline()
        if not line: # 如果这一行非空，也就是未到文末
            print(line)
```

你也可以借助 fileinput 模块，这样代码更简洁：

```
import fileinput
for line in fileinput.input(r'e:\f.txt'):
    print(line)
```

6.2.3 读取文件：列表解析法

直接上例子吧，就一行：

```
F = [line for line in open('f.txt', 'r')]
```

在 Python 的简洁面前颤抖吧，那些读文件烦到死的编程语言们。

Chapter 7

函数

终于要谈到函数了。函数用 def 语句和随其后的一个语段进行定义，在生成器这一节中已经提到过一点。

函数的普通定义很简单：

```
def func():
    """ This is a function. """
    print('Hello world')

func() # 这一行是调用
```

其中，三个双引号包裹的是利用 help(func) 能够获得的帮助字符串。对于上例，如果想要加上参数，并且给出一部分参数的默认值：

```
def func(a, b=1): # 参数a默认值是1
    return a + b  # 返回值

x = func(2) # 这个默认传入给参数a
y = func(1, 2) # 还可以写成func(a=1, b=2)
```

7.1 不定参函数

Python 支持把多个不定值的函数当作元组或者字典进行读取，比如：

```
def a_fun(x, *tup):
    ...

```

```
def b_fun(y, **dic):
    ...

a_fun(1, 2, 3, 4) # 这里(2,3,4)作为元组被传入给tup
b_fun(1, name = 'wkl', gender = 'male') # 类似。传给dic
```

7.2 全局变量

强烈建议不要在 Python 中使用全局变量，因为它显著增加了代码的阅读难度。由于我个人的厌恶，这里就不细写了。具体可以查阅 global 命令。

7.3 特殊函数

这里介绍 Python 里面的几个特殊函数，其他编程语言里面甚至可能没有类似的。另外，别忘了在这里介绍过的 zip 函数，也是很好用的。

7.3.1 lambda 函数

这个 lambda 函数有点像 MATLAB 里面的匿名函数 @(x,y,…).

```
func = lambda x, y: x + y
func(2, 5)
```

7.3.2 map 函数

map 的作用是依次对传入的数据进行操作，并返回一个列表。比如把列表中的每个数据都加 1：

```
def add_to_self(x):
    return x + 1
lst = list(map(add_to_self, range(5))) # add函数不带括号
print(lst) # 输出结果：[1, 2, 3, 4, 5]
```

对于上述写法不满意的装逼犯，请看这里：

```
lst = list(map(lambda x: x + 1, range(5)))
print(lst)
```

7.3.3 filter 函数

filter 的作用，顾名思义，是根据某一规则对数据进行“筛选”。

```
lst = list(filter(lambda x: x > 0, range(-3, 3)))
print(lst)
```

如果函数返回 True，那么对应的数据就被保留；否则对应的数据被丢弃。一个更正常的例子：

```
def func(char):
    return char != 'a'

s = filter(func, 'abcdef')
print(s)
```

7.3.4 reduce 函数

这个函数在 Python 3 里面需要加载 functools 模块。顺便一提，在 Python 2 里面，它是直接可以调用的。

```
from functools import reduce
lst = reduce(lambda x, y: x + y, range(5))
print(lst) # 结果：0 + 1 + 2 + 3 + 4 = 10
```

它的作用就是对序列中的数据，自前往后，依次用给定的函数进行处理。reduce 和 map 的区别，就好像一个是用来纵向运算、一个是用来横向运算一样。

Chapter 8

类

类（class）的成员包括两种，一种是**属性（attribute）**，一种是**方法（method）**。直接一个长例子：

```
class Person: # 类定义语句
    """ This is a claas for person info. """
    def __init____(self, personname): # 初始化方法
        self.name = personname # name是该类的一个属性

class Girl(Person): # 从Person类继承的Girl类
    def __init__(self, personname) # 重写父类方法
        super(Girl, self).__init__(personname)
        self.age = 17

class Boy(Person): # Boy类也继承Person
    @staticmethod # 这表示从这往下是静态方法
    def foo():
        print('static')

    @classmethod # 这表示从这往下是类方法
    def foooo(cls):
        print('Class{}'.format(cls.__name__))
```

现在来具体解释这段代码。

1 - 以上代码中，所有的 `self` 都是不能替换的。这里的 `self.name` 相当于 `C#` 里的 `this.name`。

2 - 三个双引号包围的字符串是帮助字符串，在函数中提及过。

3 - 关于继承，我就提醒一句：Python 3 的父类搜索方法是**广度优先**。比如类 A 和类 B 都继承于类 S，而类 C 从 A, B 进行双重继承。那么，对于一个函数，查找父类的顺序是 C、A、B、S(回溯 A 的)、S(回溯 B) 的。

4 - 在子类里面对 `__init__` 进行定义，就表明父类的方法被重写了。加上 `super()` 语句后，原来父类 `Person.__init__` 的所有语句（这里其实就一句：self.name = personname）就被加入到子类 Girl 里面了。如果不加这一句，子类 Girl 在创造实例时，就不会赋予实例 age 信息——因为 age 信息在父类中才提到。

5 - 静态方法与类方法在下一节说明。

8.1 静态方法与类方法

静态方法和类方法究竟与普通方法有什么不同呢？

普通方法： 定义中含有参数 `self`，只能先实例化，再由实例调用。

静态方法： 定义时用 `@staticmethod` 注明，无特殊参数。既可以通过类直接调用，也可以通过类的实例调用。

类方法： 定义中含有参数 `cls`，只能访问类的属性，不能访问实例的属性。

8.2 多态

函数或方法的多态性在 Python 上表现的淋漓尽致。最简单的一个运算符'+'，既可以用于数字相加，也可以用于字符串连接，这就是多态性的一个典型例子。

多态性很多时候与继承相关。一个很有名的猫与狗的例子：

```
class Animal:
    def talk(self):
        pass # 表示定义留空
class Cat(Animal): # 从Animal继承
    def talk(self): # 重写talk()
        print('Meow')
class Dog(Animal):
```

```
    def talk(self):
        print('Woof')
a, b = Cat(), Dog()
a.talk() # 'Meow'
b.talk() # 'Woof'
```

8.3 封装和“伪装”

怎么把类的成员（包括属性和方法）私有化呢？很简单粗暴，前面加两个下划线就行。

```
class Person:
    def __foo(self, personname):
        self.__name = personname
a = Person()
try: # 如果非私有，成员可以从外部直接更改值
    print(a.__name)
except: # 如果不能访问
    print('Fail to print')
    exit() # 终止整个程序
```

以上代码的运行结果就是 Fail to print. 如你所见，每个类自带的初始化方法 `__init__()` 就是一个私有化的方法。

那如果我想调用私有化的属性 `__name`，难道蠢到要写一个 `get_name()` 的方法吗？然后再 `a.get_name()` ？

那可真蠢。试试 `@property`，它能让一个**方法**“伪装”成同名**属性**，这样至少不需要输入每个方法后面都有的那个丑陋的括号了。简洁很重要。

```
class Person:
    def __init__(self, personname):
        self.__name = personname

    @property
    def name(self):
        return self.__name

a = Person('wkl')
print(a.name) # 输出结果是'wkl'
```

8.4 特殊属性和方法

这里其实可以说的有许多，我挑几个重要的说一下。

8.4.1 ___dict___ 属性

用于查看类和实例的属性与方法，类型是字典。比如一个类 Class 和它的一个实例 A，在定义类 Class 时也定义了属性 Element，那么 `Class.Element` 与 `Class.__dict__['Element']` 是等同的。

如果实例 A 的属性 Element 的值被修改过：调用 `A.Element` 时以修改后的值为准；否则就以 `Class.ELement` 的值为准。但是调用 `A.__dict__['Element]` 时，如果 `A.Element` 未被修改过，这个调用是不能成功的。

8.4.2 ___slots___ 属性

这个属性有什么用？因为非私有的类属性可以被从外部直接更改值，且属性还能在外部被增加。这个属性就是为了限制属性的滥用的。同时，它更大的意义在于优化了内存。

```
class ClassName:
    __slots__ = ('Name', 'Gender')
```

此时在输入 `dir(ClassName)`，就发现已经找不到 `__dict__` 属性了。而且现在的属性被限定为 Name 和 Gender，使得从类外部无法增加、删除、修改属性值。

8.4.3 ___getattr___ 与 ___setattr___ 方法

访问不存在的属性名称时，调用前者；赋值属性时调用后者。你可以重写它们：

```
class ClassName:
    def __getattr__(self, name):
        print('None')
    def __setattr__(self, name, value):
        self.__dict__[name] = value
```

前者实现的是在访问不存在的属性时输出字符串；后者实现的是通过 `__dict__` 属性强行使类能够在接受一个属性赋值时，即使属性不存在，也创造一个属性出来而不是报错。

如果有更多精力，可以查阅 `__getattribute__` 方法。

Chapter 9

库与模块

在前面的小节已经介绍过一点，但是我还是说几句。调用时的几种办法，我推荐第一种，同时建议你不要使用 '*' 的那种。

```
# 第一种，调用：math.pow(2, 3)
import math # 易读性最高
# 第二种，调用：pow(2,3)
from math import pow
from math import * # 这是贪心的做法
# 第三种，调用：cifang(2,3)
from math import pow as cifang
```

怎么判断一个 py 文件是 main 脚本还是一个模块呢？你可以通过在 py 文件内部的写上：

```
if __name__ == "__main__":
    <...>
```

当 py 文件作为 main 脚本执行时，上面的判断就为真；当 py 文件作为模块被 import 时，上面的判断就为假。

下面介绍一些 Python 3 的标准库的内容——也就是安装 Python 时自带的库。

9.1 sys 模块

9.1.1 sys.path

这个是必须介绍的，表示模块路径列表，Python 会从列表中搜索你想调用的模块。如果你调用的模块的路径 <ModulePath> 不在 PythonPath 中，那么：

```
import sys
if <ModulePath> not in sys.path:
    sys.path.append('<ModulePath>')
```

9.1.2 sys.stdin/stdout

Python 的标准输入输出。比如，sys.stdout 如果指定一个文件，那么 print 函数的输出结果将不会输出到控制台而是直接写入到文件里。

```
import sys
with open('f.txt', 'w') as f:
    sys.stdout = f
    print('This is a test.')
```

你可以自己试一试 sys.stdin 属性。

9.1.3 sys.exit()

这个方法用于退出程序。不含参数是会返回一个 SystemExit 错误，如果带参数 0，则表示正常退出。你也可以使用 `sys.exit('Bye')`，在退出前输出字符串到屏幕。

9.2 os 模块

os 模块用于操作文件和路径。这些事情就像你用 CMD 命令的 dir 或者 cd 一样。

9.2.1 文件操作

直接用例子说明吧：

```
import os
os.rename('old.py', 'new.py') # 重命名
os.remove('a.py') # 删除文件
os.stat('b.py') # 查看属性
```

9.2.2 路径与文件夹

同样用例子。如果参数是当前目录，一般可省略：

```
os.getcwd() # 获取当前目录
os.chdir('d:\list') # 更改当前目录为
os.chdir(os.pardir) # 返回上一级目录
os.mkdir('newfolder') # 在当前目录新建一个文件夹
os.listdir('c:\list') # 列出文件夹下所有文件的列表
os.removedirs('thefolder') # 删除空文件夹
os.stat('thefolder') # 查看目录属性，上面已介绍过
```

顺便一提，如果想删除一个非空的文件夹：

```
import shutil
shutil.retree('d:\list\thefolder')
```

9.3 re 模块：正则表达式

正则表达式对于字符串处理是非常重要的，会与不会在字符串问题上面的效率相差是成倍的。此外，正则表达式被广泛运用在整个计算机世界而不仅仅是 Python，因此阅读这部分内容也是大有裨益的。

正则表达式的工作原理是使用**元字符**（也就是.^$*+?{}[]\|()）来匹配其他字符串。这些元字符并不直接匹配自身，而是定义字符类型、子组和重复次数等等信息。

怎么知道一个正则表达式是否匹配一个字符串呢？使用 re.match() 方法。如果匹配成功它会返回一个 obj，否则它会返回 None。因此你可以用下面这个语句来判断是否匹配。

```
1 print(bool(re.match(r'\d', '3')))
```

9.3.1 匹配规则

你要记住在 Python 的字符串中，反斜杠也是转义字符。因此下面所说的比如'\d' 应该写成'\\d'，当然了我建议你使用r'\d'。以下去掉字母 r 只是为了尊重正则表达式的正常书写，请养成在 Python 中使用正则表达式时前加字母 r 的习惯。

首先看匹配单个字符：

$\backslash d/D$ 小写 d 匹配单个数字，即 0~9. 大写 D 则匹配所有非数字。比如'2\d' 能匹配'20'，却不能匹配'2Q'.

$\backslash w/W$ 小写 w 匹配单个字母或数字。大写 W 取反。

$\backslash s/S$ 小写 s 匹配空白符，包括空格、制表符、换行符等。大写 S 则取反。

. 一个英文句点，可以匹配单个任意字符（除了换行符）。如果设置了 re.DOTALL，英文句点可以匹配包括换行符在内的单个任意字符。

$ 匹配给定字符串的结束位置，也就是尾部。或者开始用\A，结束用\Z。

反斜杠 + 元字符 如果你想匹配单个元字符本身，只需要加上反斜杠。

但是这似乎匹配的范围太广泛了。如果我只想搜索 python 或 Python 这两个字符串呢。这时候需要用到中括号：

方括号 [Pp]ython 即可。如果只想匹配小写字母你也可以这样写：[a-z]，而[0-9] 与\d 实质是等价的。此外，[a-zA-Z0-9] 就相当于\w。需要注意的是，即使是元字符，在括号中也不会有别的含义，而仅仅是匹配自身。

^ 尖角符号表示取反。比如[^0-9] 表示匹配除了数字以外的任意一个字符，也就是\D。

有时需要匹配较长的字符，也就是对某一规则进行重复匹配，那就需要：

* 匹配它前面的字符 0 次到无限次。比如'ID\d* 可以匹配'ID007' 也可以匹配'ID'。再比如'h*OK' 可以匹配 hhhhhhOK。

+ 匹配它前面的字符 1 次到无限次。故'ID' 不能被'ID\d+' 匹配了。

? 匹配它前面的字符 0 次或者 1 次。

{n} 匹配它前面的字符不多不少恰好 n 次。

{n,m} 匹配它前面的字符 n 到 m 次。所以{0,} 相当于 0 到无限次，即'*'；{1,} 相当于'+'；而{0,1} 相当于'?'。注意，这里的逗号后面不可以加空格。

9.3.2 子组

另一个重要的概念是子组，也就是用圆括号包裹的内容。有一种比较有用的方法叫前向（后向）断言，用于指定匹配字符串前后的字串必须满足的条件。

(?=...) 前向肯定断言。比如 male(?=wkl)，表示只匹配紧跟着'wkl' 的字符串'male'。

(?!...) 前向否定断言。比如 male(?!wkl) 表示只匹配紧跟的内容不是'wkl' 的字符串'male'。

(?<=...) 后向肯定断言。(?<=male)wkl 表示只匹配前面紧跟内容为'male' 的字符串'wkl'。

(?<!...) 后向否定断言。与上同理。

子组可以命名，也可以有更灵活的使用方式：

\ ... 引用序号 id 对应的子组。序号从 1 开始依次编号。比如'\1'。

(?P<name>) 命名子组为 name，方便之后调用。

(?P=name) 引用一个命名过的子组。

(?:...) 非捕获组。该组的内容不能被后文引用。

(?(id/name)yes|no) 这是表示如果序号为 id 或者名字为 name 的子组匹配到目标的话，此处就尝试用 yes 表达式匹配；否则尝试用 no 表达式匹配。这个写在下面的例子里。

9.3.3 例子

一个例子：`(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` 它的用途是匹配任意电子邮件地址。假设我们要匹配的地址是 name@website.com，分析一下这个正则表达式的构成：

1 - 首先，这个正则表达式分为三块，大致是根据元组分隔的，分别是：

`(<)?` 以及`(\w+@\w+(?:\.\w+)+)` 还有`(?(1)>|$)`

2 - 先分析第一块：匹配字符 '<'0 次或 1 次。

3 - 再分析第二块，这个有点复杂。不过我们都知道按照顺序计算的道路，阅读正则表达式也一样。首先是`\w+` 这表示至少一个字母或数字，然后是`@`，这个是电子邮件地址里面都会有的。然后又是至少一个字母或数字。这样就匹配了 name@website。

`(?:\.\w+)+` 这个子组里面以`?:` 开头，表示非捕获组，不管它。然后是`\.`，这表示一个英文句点（因为它是元字符所有前加反斜杠）。然后又是至少一个字母或数字。这个就表示一个以小数点开头、后跟若干字母或数字的字符串——而这个字符串会被匹配至少一次。就好像.com 或者.com.cn 都会被匹配一样。到此为止匹配 name@website.com 或者 name@website.com.cn 都没有问题了。

4 - 最后一块，也就是上面提到的`(?(id/name)yes|no)` 语法。如果序号 1 的子组匹配成功，那么用 '>' 进行匹配，否则一直匹配到字符末尾。序号 1 的子组是什么？就是第 1 块，也就是匹配字符 '<'。所以这里就是说，如果之前匹配到了字符 '<'，那么尝试匹配字符 '>'；如果电子邮件地址开头没有字符 '<'，那么就一直匹配到字符串末尾。

5 - 结论：可以匹配 name@website.com 或者 name@website.com.cn 这样的正常电子邮件地址，也可匹配两端加了一对尖括号的电子邮件地址。如果只有一侧加了尖括号，比如 <name@website.com，就不匹配。

9.3.4 贪婪

在正则表达式的默认下，重复匹配是贪婪的，也就是匹配尽可能多的字符。比如你用`12[0-6]+` 去匹配字符串`1234564321`，它一定会返回整个字符串，而不只是找到一个'123' 就满足。

启用非贪婪模式的方法是在`.` `*` `?` 这三种字符的后面加上一个问号，变成`.?` `*?` `??`。非贪婪模式下正则表达式会尽量找到较短的结果，然后返回。

9.3.5 在 Python 中使用

你已经知道的 re.match() 方法，这里介绍编译方法 re.complie() 。因为 Python 要先确认你的正则表达式没有问题，才会拿去匹配。如果你经常需要匹配一个正则，你可以把编译结果先放在一个变量里：

```
import re
# 匹配电话号码：3-4位区号加7-8位数字
phone_re = re.compile(r'\d{3,4}-\d{7,8}')
phone_re.match('010-12345678').group() # 返回匹配结果
```

如果你想把前后两段的分别读出来，那么使用子组，并使用 groups().

```
import re
phone_re = re.compile(r'(\d{3,4})-(\d{7,8})') # 添加子组
phone_re.match('010-12345678').groups() # 返回列表
# 这里你也可以用group(0)来调用整体
# 用group(1/2)分别调用区号和电话
```

你也可以在 compile 后面加上 findall(StrName)，来找出一个字符串中所有匹配该正则表达式的子字符串。

```
import re
phone_re = re.compile(r'\d{3,4}-\d{7,8}')
phone_set = '010-12345678, 021-65439876'
lst = phone_re.findall(phone_set)
```

输出结果是：['010-12345678', '021-65439876']

还有一个很有用的指令是 re.split() ，这个可以用正则表达式来分割字符串。如果想识别连续的空格:

```
import re
s = 'a b   c'
lst = s.split(' ') # 输出：['a', 'b', '', '', 'c']
lst_re = re.split(r'\s+', s) # 输出：['a', 'b', 'c']
```

过于深入的正则表达式内容在这里就不细说了。那可以写一本书。我在此只是简单介绍。

9.4 urllib 模块

注：*urllib2* 模块在 *Python 3* 中已经被并入 *urllib.request* 。

其实这个模块也可以不提，因为涉及到一些 HTML 的知识。不过的确用 Python 爬虫是很多学习者感兴趣的事情，所以这里还是粗浅地说一下吧。。

```
import urllib.request as urllib2 # 这是照顾\python{} 2的习惯
# urllib2.Request()是向originurl请求网页的源代码
req = urllib2.Request(originurl)
# 有的网站不喜欢爬虫工具，所以你要伪装成浏览器
req.add_header('User-Agent', browser)
# urllib2.urlopen()将页面打开，read()是读源代码
byte_f = urllib2.urlopen(req).read()
# decode是将源代码以utf-8（有时GBK）编码方式解码。
    # ignore表示部分无法解码的会被跳过
utf8_f = byte_f.decode('utf-8', 'ignore')
```

以上例子是从我在个人 Github 上写的一个小脚本里面摘取的，它总共才 100 行：https://github.com/wklchris/pyCatchPic，可以参考一下。

9.5 第三方库及其他

网上的第三方库怎么安装呢？下载以后，找到里面的 setup.py，然后 cmd 切换到该路径，再输入：python setup.py install 即可。或者你可以到 UC Irvine 的这个站点下载非官方的 whl 文件，并用 pip 安装：pip install name.whl[1]

其他还有一些实用或者有趣的标准模块，可以自行尝试一下：

collection 提供了一种双端列表的数据结构 deque，可以用 appendleft, extendleft, popleft 等方法从 deque 的左侧（也就是 lst[0]）进行操作。

calendar 这个模块可以帮你获取 xx 年 x 月 x 日是周几这样的信息，以及输出一个日历到屏幕。你可以尝试：print(calendar.month(2016, 7))，当然还有很多用法。甚至判断闰年也可以。

[1]如果你的 pip 版本较低，可以尝试先运行：`python -m pip install --upgrade pip`

Chapter 10

数据读取与存储

10.1 存储与读取数据

加载 pickle 模块：

```
import pickle
lst = [1, 2, 3]
with open('1.dat', 'w') as f:
    pickle.dump(lst, f)
```

如果加上参数 True： `pickle.dump(lst, f, True)`，表示用二进制方式存储，压缩率会更高，即文件体积更小。

这样存储的数据怎么读呢？

```
with open('1.dat') as f:
    data = pickle.load(f)
```

如果需要将数据以字典的行驶存储，使用 shelve 模块。

```
import shelve
with shelve.open('1.db') as f:
    f['name'] = 'wkl'
    f['gender'] = 'male' # 新增key
```

如果你想修改已有的 key，参数是：

```
with shelve.open('1.db', writeback=True) as f:
```

10.2 连接数据库

这里只介绍一些连接数据库的 Python 库和模块。

MySQL pypi.python.org/pypi/MySQL-python

MongoDB pypi.python.org/pypi/pymongo

SQLite Python 原生支持： `import sqlite3`

Chapter 11

其他

这里主要说一些惯例，关于 Python 的变量使用较多。

1 - 命名时，变量名和函数名以**小写字母**开头，类名则以**大写字母**开头。

2 - 如果你对命名感到困难，可以参考驼峰命名法、匈牙利命名法等。

或许本手册没有太多体现，笔者的命名方法实质上是用下划线分隔的全小写命名法，有时也用不带下划线的驼峰命名法。

3 - 使用下划线的命名，不要让下划线超过 3 个。最好也别超过 2 个。

4 - 一些 bool 类型的变量或返回同类型值的方法，命名建议用 is 或者 has 开头，这样能够迅速知道它是布尔型的。比如 is_open 和 has_gender.

5 - 虽然 Python 中一个变量可以被赋给不同的数据类型，但**永远不要在代码中这样做**。

6 - 如果能不用某个临时变量，同时又不会把某一行代码嵌套的很难懂，那请你不要使用这个临时变量。不要说“我觉得**可能**会在后面用到，所以创建个变量吧”，除非你“肯定”。

7 - 不要在一开始就交待所有的变量赋值，然后过很多行再拿出来用。记住，“就近赋值”。

8 - 你也可以尝试在大程序中，强迫自己在每一阶段的代码后面删除掉在下一阶段不会使用到的变量。这一条带来的可读性上升建立在前两条的基础上。

9 - 注释不要太多，更重要的是变量命名易懂。注释即抱歉。

10 - 空行不要太多也不宜太少。如果你用 PyCharm，它会用近乎偏执的方法不停提醒你更正格式。事实上 Python 本来就是一门注重格式的编程语言，多数情况下听 PyCharm 的挺好。

最后拿什么结尾呢？当然是《Python 之禅》了。这首……姑且称为诗的东西，利用 `import this` 这条语句就可以看到。原文没有分节，嗯，我就自作主张分个节好了。

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.

Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.

Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

以上字体叫做 *Lucida Calligrpahy*，是我最喜欢的西文字体之一。

全 手 册 完