Python 编程

杨志宏

修订于：2019-12-21

献给可爱的同学们，是你们求（wú）知（jīng）若（dǎ）渴（cǎi）的眼神激（cì）励（jī）着我！

目录

表格

插图

Python 简介

在这本电子书中，我们将学习 Python 的基础知识，最终达到抓取网络数据、分析数据的目的。

Life is short, you need Python。Bruce Eckel

0.1 Python 发展历史

Python 的创始人为 Guido van Rossum。1989 年圣诞节期间，在阿姆斯特丹，Guido 为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，做为 ABC 语言的一种继承。之所以选中 Python（大蟒蛇的意思）作为程序的名字，是因为他是一个叫 Monty Python 的喜剧团体的爱好者。ABC 是由 Guido 参加设计的一种教学语言。就 Guido 本人看来，ABC 这种语言非常优美和强大，是专门为非专业程序员设计的。但是 ABC 语言并没有成功，究其原因，Guido 认为是非开放造成的。Guido 决心在 Python 中避免这一错误。同时，他还想实现在 ABC 中闪现过但未曾实现的东西。

截至目前，Python 的版本为 3.7.4，2019 年 7 月 8 日发布。

0.2 Python 特点

1. 简单 Python 是一种代表简单主义思想的语言。阅读一个良好的 Python 程序就感觉像是在读英语一样。它使你能够专注于解决问题而不是去搞明白语言本身。
2. 易学 Python 很容易上手，一方面是由于 Python 有完善的说明文档，另一方面网络中有大量的教程，学习资源可谓丰富。本书的写作就参考了诸多网络教程。\cite{pythonguru, 2015}
3. 开源开源意味着人们可以自由地发布这个软件的拷贝、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。Python 有非常活跃的开源社区，来自世界各地的程序员不断完善着 Python，如今 Python 拥有功能强大且门类齐全的扩展库。它可以帮助处理各种工作，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV 文件、密码系统、GUI（图形用户界面）、Tk 和其他与系统有关的操作。Python 语言及其众多的扩展库所构成的开发环境十分适合工程技术、科研人员处理实验数据、制作图表，甚至开发科学计算应用程序。
4. 解释性在计算机内部，Python 解释器把源代码转换成称为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。这使得使用 Python 更加简单。也使得 Python

程序更加易于移植。

5. 可移植 Python 已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。这些平台包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE、PocketPC、Symbian 以及 Google 基于 linux 开发的 android 平台。
6. 面向对象 Python 既支持面向过程的编程也支持面向对象的编程。在“面向过程”的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在“面向对象”的语言中，程序是由数据和功能组合而成的对象构建起来的。
7. 可扩展如果需要一段关键代码运行得更快或者希望某些算法不公开，可以部分程序用 C 或 C++ 编写，然后在 Python 程序中使用它们。
8. 可嵌入可以把 Python 嵌入 C/C++ 程序，从而向程序用户提供脚本功能。

0.3 使用 Python 的知名项目

以下是使用 Python 作为主力开发语言的知名项目，其中有一些是用 python 进行开发，有一些在部分业务或功能上使用到了 python，还有的是支持 python 作为扩展脚本语言。

1. Reddit 社交分享网站，最早用 Lisp 开发，在 2005 年转为 python。
2. Dropbox 文件分享服务。
3. 豆瓣网图书、唱片、电影等文化产品的资料数据库网站。
4. Django 鼓励快速开发的 Web 应用框架。
5. EVE 网络游戏 EVE 大量使用 Python 进行开发。
6. Fabric 用于管理成百上千台 Linux 主机的程序库。
7. Blender 以 C 与 Python 开发的开源 3D 绘图软件。
8. BitTorrent bt 下载软件客户端。
9. Ubuntu Software Center Ubuntu 9.10 版本后自带的图形化包管理器。
10. YUM 用于 RPM 兼容的 Linux 系统上的包管理器。
11. Civilization IV 游戏《文明 4》。
12. Battlefield 2 游戏《战地 2》。
13. Google 谷歌在很多项目中用 python 作为网络应用的后端，如 Google Groups、Gmail、Google Maps。
14. NASA 美国宇航局，从 1994 年起把 python 作为主要开发语言。
15. Industrial Light & Magic 工业光魔，乔治 · 卢卡斯创立的电影特效公司。
16. Yahoo Groups 雅虎推出的群组交流平台。
17. YouTube 视频分享网站，在某些功能上使用到 python。
18. Cinema 4D 一套整合 3D 模型、动画与绘图的高级三维绘图软件，以其高速的运算和强大的渲染插件著称。
19. Autodesk Maya 3D 建模软件，支持 python 作为脚本语言。
20. gedit Linux 平台的文本编辑器。
21. GIMP Linux 平台的图像处理软件。
22. Minecraft: Pi Edition 游戏《Minecraft》的树莓派版本。

23. MySQL Workbench 可视化数据库管理工具。
24. Digg 社交新闻分享网站。
25. Mozilla 为支持和领导开源的 Mozilla 项目而设立的一个非营利组织。
26. Quora 社交问答网站。
27. Path 私密社交应用。
28. Pinterest 图片社交分享网站。
29. SlideShare 幻灯片存储、展示、分享的网站。
30. Yelp 美国商户点评网站。
31. Slide 社交游戏/应用开发公司，被谷歌收购。

0.4 学习资源

1. 简明 Python 教程[1]
2. 官方文档[2]

[1] https://bop.molun.net/

[2] https://www.python.org/doc/

Python 开发环境搭建

0.5 在 Windows 中安装 Python 3

Python 支持多个平台，其中在 Mac、类 UNIX 平台中已默认安装，Windows 平台中的安装也非常简单，从官方网站下载安装包安装即可，注意安装完成后将 Python 所在目录添加到系统路径（PATH）中。

0.6 在 MacOS 中安装 Python 3

在目前的 MacOS 中，内置的 Python 版本为 2.7。如果要安装 Python3，步骤如下：

0.6.1 安装 Xcode

在终端中运行如下命令：

```
xcode-select --install
```

0.6.2 安装 Homebrew

在终端中运行如下命令：

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Homebrew 安装完后，使用 vim 或者其他编辑器将如下内容添加到 `~/.profile` 文件中。

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

0.6.3 安装 Python3

在终端中运行如下指令：

```
brew install python3
```

0.6.4 使用 Python3

使用 Homebrew 安装 Python3 后，系统就有两个 Python 环境，如果要使用系统自带的 Python2 ，则使用如下指令：

```
python
```

而如果要使用 Python3，则使用如下指令：

```
python3
```

0.7 使用 python 虚拟环境

0.7.1 为什么要用虚拟环境

虽然使用 python3 这样的工具，能够使用 python3 解释器，但在开发人员需要在不同的版本中安装扩展包时，或者在需要使用不同版本的某个解释器、扩展包时，如果只使用系统唯一的开发环境，就显得捉襟见肘。

0.7.2 虚拟环境的创建与使用

因此，Python 提供了创建虚拟环境的工具 `venv`，例如：

```
python3.7 -m venv python37
```

上面的命令将在当前目录中创建名为 `python37` 的目录，其中参数`-m` 的作用是让后面的库以脚本的形式运行（-mod）。进入该目录后，再运行：

```
source bin/activate
```

就会激活该虚拟环境，命令后会出现（`python37`）这样的提示符。

在该目录中运行 `deactivate` 则会退出虚拟环境。

0.7.3 虚拟环境与系统环境的区别

使用 venv 命令创建的虚拟环境是一个独立于系统 Python 目录的轻量级 Python 目录，虚拟环境目录和系统 Python 目录之间共享的是 Python 标准库，而每个虚拟目录各自拥有独立的 Python 扩展以及各自的 pip 包管理，如果系统有多个版本的 Python 主库，也可以基于其自身创建不同版本的 Python 虚拟环境。

在一个虚拟环境中，只能有一个版本的 Python。而在系统环境中，可以有多个版本的 Pyhton。

0.8 编辑器

虽然 Python 自带编辑器，但其不够方便，推荐使用轻量级的编辑器 Sublime Text 或者 Visual Studio Code。使用编辑器将文件保存成 `.py` 后缀，然后通过命令行调用即可执行。

如在编辑器中键入如下内容：

```
print("hello world")
```

保存为 `hello.py`（文件名不能与 Python 的各种函数、库名相同），注意设置文件编码方式为 `UTF-8`。

启动终端，进入到脚本所在路径，执行：

```
python hello.py
```

即可看到运行结果。

0.8.1 Visual Studio Code 中的必要设置

使用 Visual Studio Code 作为 Python 开发编辑器，最好进行如下操作：

1. 安装 Python 扩展。
2. 安装语法提示插件。
3. 选择解释器。当 Visual Studio Code 遇到 Python 虚拟环境的时候时常会无法找到正确的 Python 解释器和虚拟环境，导致调试无法进行。解决的方案是：选择 Visual Studio Code 的设置菜单，在工作目录（workspace）中，正确设置 `python.pythonPath` 的值，这样可以避免项目之间发生版本冲突。

第一部分

核心语法

核心语法

以下对 Python 的核心语法进行介绍。

0.9 注释

在 Python 中，使用 # 标记注释。注释不会被 Python 解释器执行。注释是开发人员用来提醒自己或他人程序如何工作的重要手段，注释还会用在文档的写作中。

```
# -*- coding: utf-8 -*-
# 注释不会运行
print('hello world')
```

上述代码将会打印出 hello world 字符串。

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。

所谓物理行（Physical Line）是你在编写程序时你所看到的内容。所谓逻辑行（Logical Line）是 Python 所看到的单个语句。Python 会假定每一物理行会对应一个逻辑行。有关逻辑行的一个例子是诸如 print('hello world') 这样的一行语句，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。

0.10 变量

变量（Vaiable）实质上是对内存中地址的命名，在内存中存储着诸多对象，为了方便使用这些对象，便有了变量。把变量和函数的名称我们叫作标识符（Identifier）。

0.10.1 变量名称

在 Python 中，标识符必须遵守以下规则：

1. 所有标识符必须以字母或者下划线 (_) 开头，不能以数字开头。如 my_var 就是一个有效的标识符，而 1digit 就不是。

2. 标识符可以包含字母、数字和下划线。标识符不限长度。
3. 标识符不能是关键字（所谓关键字，就是 Python 中已经使用并有特定含义的单词）。Python 的关键字参见附录 Python 关键字。

0.10.2 变量赋值

值（Value）是程序运行过程中的基本元素之一，例如 1，3.14，“hello” 等等都是值。在编程属于中，它们又被叫作字面量（literals）。字面量拥有不同的类型，如 1 是整型（int），3.14 是浮点型（float），“hello” 是字符串（string）。

在 Python 中，无需声明变量类型，解释器会根据变量的值自动判断变量类型。使用等于号为变量赋值，等于号也被认为赋值操作符（operator）。以下是变量声明的一些例子：

```
x = 100                         # x 是整型
pi = 3.14                       # pi 是浮点类型
empname = "python is great"     # empname 是字符串
a = b = c = 100                 # 将100赋值给a、b、c
```

注意，变量 x 中并不储存 100 自身，它存储的是 100（它是一个整型对象）的引用（reference）地址。

0.10.3 同步赋值

Python 可以使用以下语法对多个变量同步赋值：

```
var1, var2, ..., varn = exp1, exp2, ..., expn
```

上述声明告诉 Python，将表达式右边的值依次赋值给表达式左侧的变量。同步赋值在要交换两个变量的值时非常有用。例如：

```
x = 1
y = 2

y, x = x, y # 交换x、y的值

print(x)
2
print(y)
1
```

0.11 数字类型

Python 3 支持 3 种不同类型的数字类型。

1. `int` 整型数字，比如 `2015`。
2. `float` 浮点型数字，比如 `3.14`。
3. `complex` 复数，比如 `3+2j`。

0.11.1 查看变量类型

Python 使用内置函数 `type()` 来查看变量的类型。在 Python 中，内置了一些高效强大的对象类型，使得开发人员不用从零开始进行编程。实际上，Python 中的每样东西都是对象。虽然 Python 中没有类型声明，但表达式的语法决定了创建和使用的对象的类型。一旦创建了一个对象，它就和操作集合绑定了，这就是所谓的动态类型和强类型语言。即 Python 自动根据表达式创建类型，一旦创建成功，只能对一个对象进行适合该类型的有效操作。

```
>>> x = 12
>>> type(x)
 <class 'int'>
```

0.11.2 整型

整型（int）字面量在 Python 中属于 `int` 类。

```
>>> i = 100
>>> i
100
```

数字可以进行各种运算，如：

```
123 + 345
```

还可以使用数学模块进行更高级的运算，如产生随机数等等：

```
import random
print(random.random())
```

`import` 表示引入模块，`import random` 就是引入随机数模块。

0.11.3 浮点类型

浮点数（float）是指有小数点的数字。

```
>>> f = 12.3
>>> type(f)
<class 'float'>
```

0.11.4 复数

复数（Complex number）由实数和虚数两部分构成，虚数用 j 表示。我们可以这样定义一个复数：

```
>>> x = 2+3j
>>> type(x)
<class 'complex'>
```

0.12 运算符

Python 有各种运算符，我们可以使用这些运算符完成计算。运算符见下表：

名称	含义	例子	运行结果
+	加	3+1	4
-	减	40-2	38
*	乘	3*2	6
/	除	6/3	2
//	取整除	3//2	1
	幂	2**3	8
%	求余数	7%2	1

0.12.1 运算符的优先级别

Python 按照运算符的有限级别计算表达式的值，比如：

```
>>> 3 * 4 + 1
```

在上面的表达式中，应该先进行加运算还是乘运算？为了搞清楚这个问题，我们需要明白 Python 中运算符的优先级别，表**??**显示了运算符的优先级别，依次从高到底排列如下：

运算符	描述
‘expression,...’	字符串转换
{key:datum,...}	字典显示
{[expression,...]}	列表显示

运算符	描述
()	分组
f(args...)	函数调用
x[index:index]	列表切分
x[index]	元素下标
x.attr	调用对象属性
**	指数运算
{^x}	按位取反
+x,-x	正负号
*，/，%	乘、除、取余数
+，-	加，减
«，»	逐位左移，逐位右移
&	逐位求和
^	逐位异或
|	逐位或
<,<=,>,>=,<>,!=,==	比较
is,not is	同一性测试
in,not in	成员资格判断
not x	布尔“非”
and	布尔“并”
or	布尔“或”
lambda	Lamada 表达式

在上表中我们可以看到，乘法运算的级别高于加法，因此，先进行乘法运算，再进行加法运算，最后的计算结果为 13。

```
>>> 3 * 4 + 1
>>> 13
```

让我们再看下面的例子，以便演示优先顺序的另一个问题：

```
>>> 3 + 4 - 2
```

上述表达式到底先进行加法运算还是减法呢？因为在运算符的优先级别表中我们看到加减运算的优先级别相同。当优先级别相同时，表达式从左向右计算，也就是说，上述的例子将先进行加法运算，再进行减法运算。

```
>>> 3 + 4 - 2
>>> 5
```

同级别运算符从左到右运算，这条规则有个例外，那就是赋值运算（=），赋值运算是从右向左计算的。例如：

```
a = b = c
```

先将 c 的值，赋给 b，再将 b 的值赋给 a。

0.12.2 增强赋值运算符

增强赋值运算符能简化赋值声明语句，例如：

```
>>> count = 1
>>> count = count + 1
>>> count
2
```

使用增强赋值运算符，我们可以将上述代码变为：

```
>>> count = 1
>>> count += 1
>>> count
2
```

类似的增强赋值运算符，除了 `+=` 外，还有`-=`，`%=`，`/=` 等等。

0.13 序列

序列（Sequence）是一个包含其他对象的有序集合，序列中的元素包含了一个从左到右的顺序，可以根据元素所在的位置进行存储和读取。Python 中内建了 6 种序列，分别是列表、元组、字符串、unicode 字符串、buffer 对象和 xrange 对象。

序列作为 Python 的数据结构，有一些操作是通用的，如：索引、分片、加、乘以及检查某个成员是否属于序列的成员（成员资格），另外，还有一些计算长度、找到最大元素等等的内建函数。

0.13.1 索引

序列中的所有元素都有编号，从 0 开始，可以按照编号来访问序列中的元素，这个标号就是索引 (indexing)。

```
se = 'Hello'
print(se[0])
print(se[-1])
```

se[0] 表示序列 se 中的第一个元素，se[-1] 表示序列中的最后一个元素。

0.13.2 分片

分片（Slicing）操作指的是访问序列中一定范围之内的元素。分片通过冒号相隔的两个索引来实现，第一个索引是需要提取部分的第 1 个元素的编号，而第二个索引是分片之后剩下部分的第 1 个元素的编号，第二个索引不包含在分片之中：

```
se = 'Hello Pythoner! '
print(se[0:5])
```

上述代码将打印出 'Hello' 字符串。但有时，我们需要获取序列的后面几个元素，同时，序列的大小是未知的，我们可以这样写：

```
se = 'Hello Pythoner! '
print(se[-9:])
```

se[-9:] 中空了第 2 个索引，表示一直到最后一个元素。上述代码将打印出 'Pythoner! ' 字符串。

进行分片时，分片的开始和结束点需要指定。而另外一个参数步长（step length）通常默认为 1，当有必要时，可是指定切片的步长，如每隔 1 个元素就取出元素：

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[0:10:2])
print(numbers[1::2])
```

上述代码将打印出 '[1, 3, 5, 7, 9]' 和 '[2, 4, 6, 8, 10]'，其中的步长都是 2。当然步长也可以设置为负值，这样分片会从后往前进行。

0.13.3 序列相加

可以通过加号能对两个相同类型的序列进行连接运算，如字符串：

```
hello = '你好'
name = 'yangjh'
print(hello + name)
```

上述代码将打印出 '你好 yangjh' 字符串。

0.13.4 序列相乘

序列乘以数字，表示将原有序列重复若干次：

```
hello = '你好'
print(hello * 3)
```

上述代码将打印出‘你好你好你好’字符串。

空列表可以使用‘[]’来表示，但是，如果想创建有 10 个空元素组成的列表，就需要使用 None，None 是 Python 内建的一个值，表示什么都没有，因此，要创建含有 10 个空元素的列表，就可以这样：

```
print([None] * 10)
```

0.13.5 成员资格

使用 in 运算符，可以检查某个元素是否存在与指定的序列中。如果元素存在于序列中，则返回 True，否则返回 False。

```
print('张三' in ['张三', '李四', '王二'])
```

上述代码将打印出布尔值 True。

0.13.6 长度、最小值、最大值

dir() 函数可以输出对象的内置方法。如：dir(‘str’) 就可以打印出所有字符串对象的内置方法。

内建函数 len() 可以返回序列的大小，如：

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(len(numbers))
print(max(numbers))
print(min(numbers))
```

上述代码将打印出 `numbers` 序列的长度 `10` 和最大值以及最小值。

0.14 字符串

Python 中的字符串（Strings）是用单引号或双引号标记的一系列连续字符（characters），换句话说，字符串是由单个字符组成的序列（list）。即便只有一个字符，也是字符串，Python

中没有字符数据类型。记住单引号括起的字符串和双引号括起的字符串是一样的，它们不存在任何区别。

0.14.1 创建字符串

```
>>> name = "tom"
>>> mychar = 'a'
```

我们还可以使用下面的语法创建字符串：

```
>>> name1 = str() # 创建一个空字符串
>>> name2 = str("newstring") # 创建一个内容为 newstring 的字符串
```

0.14.2 字符串的不可变性

在 Python 中，每一个对象都可以分为不可变性或者可变性。在核心类型中，数字、字符串和元组是不可变的。

字符串在 Python 中一旦创建就不能就地改变，例如不能通过对其某一位置进行赋值而改变字符串。下面的语句就会出现如下语法错误：“TypeError: ‘str’ object does not support item assignment”。

```
s = 'string'
print(len(s))
print(s[0])                    # 输出序列的第一个元素
s[0] = 'another s'             # 试图修改字符串的内容
print(s)
```

关于不可变性，我们再看一个例子：

```
>>> str1 = "welcome"
>>> str2 = "welcome"
```

上述代码中，str1 和 str2 都指向存储在内存中某个地方的字符串对象 “welcome”。我们可以通过 id() 函数来测试 str1 和 str2 是否真的指向同一个对象。

`id()` 函数可以得到对象在内存中的存储地址。

如下：

```
>>> str1 = 'welcome'
>>> str2 = 'welcome'
>>> id(str1)
35462112
>>> id(str2)
35462112
```

我们可以看到，str1 和 str2 都指向同一个内存地址，因此，他们都指向同样的对象“welcome”。下面让我们再编辑 str1 的内容看看：

```
>>> str1 += " yangjh"
>>> str1
'welcome yangjh'
>>> id(str1)
35487600
```

我们可以看到，现在变量 str1 指向了一个完全不同的内容地址，这也说明，我们对 str1 的内容操作实际上是新建了一个新的字符串对象。

0.14.3 字符串操作

字符串索引开始于 0，因此，我们可以这样获取字符串的第一个字符：

```
>>> name = 'yangjh'
>>> name[0]
'y'
```

在对字符串操作时，还可以从后往前取元素：

```
>>> name[-1]
'h'
```

运算符“+”用来连接字符串，运算符“*”用来重复字符串，例如：

```
>>> s = "tom and " + "jerry"
>>> print(s)
tom and jerry
>>> s = "love " * 3
>>> print(s)
love love love
```

0.14.4 字符串分片

我们还可以通过“[]”操作符来获取原始字符串的子集，这就是所谓的分片。语法规则如下：

```
s[start:end]
```

切分操作将返回字符串的部分内容，起始于 index，结束于 end-1。例如：

```
>>> s = 'yangjh'
>>> s[1:3]
'an'
>>> s = "Welcome"
>>> s[ : 6]
'Welcom'
>>> s[4 : ]
'ome'
>>> s[1 : -1]
'elcom'
```

注意：开始索引和结束索引都是可选的，如果忽略，开始索引就是 0，而结束索引就是字符串的最后一个字符对应的索引值。

0.14.5 in 和 not in 操作符

我们可以使用 in 和 not in 操作符检查一个字符串是否存在于另一个字符串，in 和 not in 就是所谓的成员资格操作符（membership operator）。

```
>>> s1 = "Welcome"
>>> "come" in s1
True
>>> "come" not in s1
False
```

0.14.6 String 对象的方法

下表是三个常用的字符串方法：

方法名称	功能描述
len()	返回字符串长度
max()	返回字符串中 ASCII 编码值最大的字符
min()	返回字符串中 ASCII 编码值最小的字符

```
>>> len("hello")
5
>>> max("abc")
'c'
>>> min("abc")
'a'
```

0.14.7 比较字符串

我们可以使用（ > , < , <= , <= , == , != ）比较两个字符串。Python 比较字符串是按照编纂字典的方式进行的，也就是使用 ASCII 编码值[3][4]

假设 str1 的值为 “Jane”，str2 的值为 “Jake”，首先比较这两个字符串的第一个字符 “J”，如果相等，就继续比较第二个字符（a 和 a），因为相同，所以继续比较第三个字符（n 和 k），因为 n 的 ASCII 编码值大于 k，因此 str1 大于 str2。更多例子参见下面的代码：

```
>>> "tim" == "tie"
False
>>> "free" != "freedom"
True
>>> "arrow" > "aron"
True
>>> "green" >= "glow"
True
>>> "green" < "glow"
False
>>> "green" <= "glow"
False
>>> "ab" <= "abc"
True
```

0.14.8 遍历字符串

字符串是序列，因此也可以使用循环遍历成员。

```
>>> s = "yangjh"
>>> for i in s:
```

[3] http://tool.oschina.net/commons?type=4

[4] 美国信息交换标准码 (American Standard Code for Information Interchange) 是由美国国家标准学会 (American National Standard Institute,ANSI) 制定的单字节字符编码方案，供不同计算机在相互通信时用作共同遵守的西文字符编码标准，它已被国际标准化组织（ISO）定为国际标准，称为 ISO646 标准。比较字符。

```
...     print(i, end="")
...
yangjh
```

改变 print() 函数的输出格式

print() 函数在默认状态下，会另起一行打印字符串，我们可以使用第二个参数修改结束标记。如 print("my string", end="") 就表示打印字符串，但不另起一行。

0.14.9 字符串内容检验

Python 字符串类内置了丰富的方法，使用这些方法（见表、ref{tab: 字符串内容检验}），我们可以检查字符串内容的类型。

方法名称	方法说明
isalnum()	如果 str 包含字符都是字母或数字则返回 True
isalpha()	如果 string 包含字符都是字母则返回 True
isdigit()	如果 string 包含字符都是数字则返回 True
isidentifier()	判断字符串是否是合格的标识名
islower()	判断字符串中是否都是小写字母
isupper()	判断字符串中是否都是大写字母
isspace()	判断字符串是否由空格组成

这些判断方法的实例如下：

```
>>> s = "welcome to python"
>>> s.isalnum()
False
>>> "Welcome".isalpha()
True
>>> "2012".isdigit()
True
>>> "first Number".isidentifier()
False
>>> s.islower()
True
>>> "WELCOME".isupper()
True
>>> "  \t".isspace()
True
```

0.14.10 在字符串内查找和替换

除了一般的序列操作，字符串还有独有的一些方法。如查找和替换：

```
print(s.find('in'))
print(s.replace('g', 'gs'))  # 虽然显示字符串已被替换，但实际上是一个新的字符串。
```

相关的方法见下表：

方法名称	方法说明
endswith(s1: str): bool	如果字符串以指定的字符串结尾，则返回真
startswith(s1: str): bool	如果字符串以指定的字符串开始，则返回真
count(substring): int	返回子字符串在字符串中出现的次数
find(s1): int	返回子字符串在字符串中第一次出现的索引，如果没有，则返回-1
rfind(s1): int	返回子字符串在字符串中最后一次出现的索引，如果没有，则返回-1

示例如下：

```
>>> s = "welcome to python"
>>> s.endswith("thon")
True
>>> s.startswith("good")
False
>>> s.find("come")
3
>>> s.find("become")
-1
>>> s.rfind("o")
15
>>> s.count("o")
3
```

0.15 列表

Python 的列表（list）对象是最常用的序列 (Sequence)。与字符串是不可变序列不同，列表是可变的。可通过对偏移量进行修改和读取。

0.15.1 列表赋值

列表可通过索引对其对应的元素进行赋值，从而改变列表的内容，如：

```
>>> a = [2, 2, 2]
>>> a[1] = 1
>>> print(a)
[2, 1, 2]
```

通过上述代码的运行，我们可以看到列表确实是可以改变的。

0.15.2 删除元素

使用 del 语句可以删除列表中的元素，如：

```
>>> a = [2, 2, 2]
>>> del a[1]
>>> print(a)
[2, 2]
```

0.15.3 分片赋值

分片赋值可以一次为多个元素赋值，并且不用考虑原列表的长度是否和新的列表长度一直，如：

```
>>> name = list('Python')
>>> print(name)
['P', 'y', 't', 'h', 'o', 'n']
>>> name[2:] = list('data')
>>> print(name)
['P', 'y', 'd', 'a', 't', 'a']
```

上述代码中的 list 函数是 Python 内置函数，其作用是将字符串转换为列表。运行结果显示，通过分片赋值，将原有列表 [‘P’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]，修改为 [‘P’, ‘y’, ‘d’, ‘a’, ‘t’, ‘a’]。

分片赋值还可以用来插入元素，如：

```
>>> name = list('Python')
>>> name[1:1] = list('--')
>>> print(name)
['P', '-', '-', 'y', 't', 'h', 'o', 'n']
```

结果显示将原有列表 [‘P’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]，修改为 [‘P’, ‘-’, ‘-’, ‘y’, ‘d’, ‘a’, ‘t’, ‘a’]。

0.15.4 列表对象常用内置方法

0.15.4.1 追加列表元素

。列表提供了在列表尾部追加新对象的方法 append。

```
>>> code = [1, 2, 3]
>>> code.append(4)
>>> print(code)
[1, 2, 3, 4]
```

0.15.4.2 计数

count 方法统计指定元素在列表中出现的次数，如：

```
>>> code = ['to', 'be', 'or', 'not', 'to', 'be']
>>> print(code.count('to'))
2
```

以上代码将统计出列表中 'to' 元素出现的次数，结果为 2。

0.15.4.3 合并列表

extend 方法在列表的末尾一次性追加另一个序列中的多个值，如：

```
a = [1, 2, 3]
b = [4, 5, 6]
a.extend(b)
print(a)
```

以上代码将把 b 列表追加到 a 列表中，打印出的 a 列表的值为 [1, 2, 3, 4, 5, 6]。和序列加运算不同，extend 方法将改变原有列表的内容，而加运算却不会。例如：

```
b = [4, 5, 6]
b + [7, 8, 9]
print(b)
```

上述代码结果显示为 [4, 5, 6]，b 列表的内容并没有改变。

0.15.4.4 元素索引

index 方法用于从列表中找出指定值第一次匹配的索引值。例如：

```
a = [1, 2, 3, 3, 2, 1]
print(a.index(1))
```

以上代码运行结果为 0，即第一个 1 出现的索引为 0。

0.15.4.5 插入元素

insert 方法用于将对象插入到列表中，例如：

```
a = [1, 2, 3]
a.insert(2, 2.5)
print(a)
```

运行结果为 [1, 2, 2.5, 3]，insert 方法的两个参数值很好理解，第一个参数为在哪个元素后插入，表示位置，第二个参数为插入的内容。

0.15.4.6 pop

pop 方法会移除列表中的一个元素，默认为最后一个，和 append 方法刚好相反，并且返回该元素的值。例如：

```
a = [1, 2, 3]
print(a.pop())
print(a)
```

运行结果为 3 和 [1, 2]，当然，pop 方法也可以指定移除某个索引的元素。

0.15.4.7 remove

remove 方法用于移除列表中某个值的第一个匹配项：

```
code = ['to', 'be', 'or', 'not', 'to', 'be']
print(code.remove('or'))
print(code)
```

运行结果为 None 和 [‘to’, ‘be’, ‘not’, ‘to’, ‘be’]。这说明 remove 方法并不返回匹配到的内容。

0.15.4.8 reverse

reverse 方法将倒序排列列表元素：

```
a = [1, 2, 3]
a.reverse()
print(a)
```

运行结果为 [3, 2, 1]。

0.15.4.9 sort

sort 方法用于对列表排序，如：

```
a = [1, 3, 4, 8, 6, 2]
a.sort()
print(a)
```

运行结果为：[1, 2, 3, 4, 6, 8]。需要注意的是，sort 方法没有返回值，并且改变列表的内容，如果你不但要排序，而且还要保持原有数据的内容，解决的方法之一是将原有内容赋值到另外一个变量中保存。

0.16 字典

字典（Dictionary）是 Python 中的一种数据类型，用来存储键（key）值（value）对。字典数据能够使用键名快速取回、添加、删除、编辑值。字典和其他语言中的数组（array）或者哈希表（hash）非常相似。字典是可变（mutable）序列。

0.16.1 创建字典

使用花括弧 {} 就可创建字典。字典中的每一个项目都由键名、冒号：和值组成，多个项目之间用逗号，分割。让我们看一个实例：

```
friends = {
'tom'   : '66666666',
'jerry' : '88888888'
}
```

上面的变量 friends 是一个含有两个项目的字典。需要注意的一点是，键名必须是可哈希类型，而值可以是任意类型。字典中的键名必须是唯一的。

0.16.2 获取、修改和添加字典元素

获取字典中的项目，使用如下语法：

```
dictionary_name['key']
```

例如：

```
>>> friends['tom']
'66666666'
```

如果字典中存在指定的键名，则返回对应的值，否则抛出键名异常。

添加和编辑项目，使用如下语法：

```
dictionary_name['newkey'] = 'newvalue'
```

例如：

```
>>> friends['bob'] = '99999999'
>>> friends
{'jerry': '88888888', 'bob': '99999999', 'tom': '66666666'}
```

删除字典中的项目使用如下语法：

```
del dictionary_name['key']
```

例如：

```
>>>  del friends['bob']
>>>  friends
{'tom': '66666666', 'jerry': '88888888'}
```

0.16.3 遍历字典

我们可以使用循环遍历字典中的所有项目。

```
>>> friends = {
     'tom'   : '66666666',
     'jerry': '88888888'
 }
>>> for key in friends:
     print(key, ":", friends[key])
```

```
tom : 66666666
jerry : 88888888
```

0.16.3.1 字典比较

使用 == 和 != 操作符判断字典是否包含相同的项目。

```
>>> d1 = {"mike":41, "bob":3}
>>> d2 = {"bob":3, "mike":41}
>>> d1 == d2
True
>>> d1 != d2
False
>>>
```

不能使用其它的关系操作符（< , > , >= , <= ）比较字典类型变量。

0.16.4 字典常用方法

Python 提供了多个内置的方法，用来操作字典，常用方法见下表：

方法名	方法用途
popitem()	返回并移除字典中的任意项目
clear()	删除字典中的所有项目
keys()	以元组的形式获得字典的键名
values()	以元组的形式获得字典的值
get(key)	获得指定键名对应的值
pop(key)	移除指定键名的项目

```
>>> friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}

>>> friends.popitem()
('tom', '111-222-333')

>>> friends.clear()

>>>  friends
{}
```

```
>>> friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}

>>> friends.keys()
dict_keys(['tom', 'bob', 'jerry'])

>>> friends.values()
dict_values(['111-222-333', '888-999-666', '666-33-111'])

>>> friends.get('tom')
'111-222-333'

>>> friends.get('mike', 'Not Exists')
'Not Exists'

>>> friends.pop('bob')
'888-999-666'

>>> friends
{'tom': '111-222-333', 'jerry': '666-33-111'}
```

0.16.5 字典的排序

字典的排序，可以使用 sorted() 函数，语法如下：

```
sorted(iterable,key,reverse)
```

1. iterable 表示可以迭代的对象，例如可以是 dict.items()、dict.keys() 等;
2. key 是一个函数，用来选取参与比较的元素;
3. reverse 则是用来指定排序是倒序还是顺序，reverse=true 则是倒序，reverse=false 时则是顺序，默认时 reverse=false。

0.16.5.1 sorted 函数按 key 值对字典排序

直接使用 sorted(d.keys()) 就能按 key 值对字典排序，这里是按照顺序对 key 值排序的，如果想按照倒序排序的话，则只要将 reverse 置为 true 即可。

```
>>> dd = {'borisakunin': 4691, 'doctor_liza': 3046, 'tareeva': 2970, 'cheger': 2887, 'kari
>>> sorted(dd.keys())
['borisakunin', 'cheger', 'doctor_liza', 'elladkin', 'karial', 'masha_koroleva', 'samoleg'
```

```
>>> sorted(dd.keys(),reverse=True)
['tareeva', 'snorapp', 'samoleg', 'masha_koroleva', 'karial', 'elladkin', 'doctor_liza', 'chege
```

0.16.5.2 sorted 函数按 value 值对字典排序

要对字典的 value 排序则需要用到 key 参数，常使用 lambda 表达式的方法，如下：

```
>>> sorted(dd.items(),key=lambda item:item[1])
[('samoleg', 2597), ('elladkin', 2616), ('masha_koroleva', 2683), ('snorapp', 2707), ('karial',

>>> sorted(dd.items(),key=lambda item:item[1],reverse=True)
[('borisakunin', 4691), ('doctor_liza', 3046), ('tareeva', 2970), ('cheger', 2887), ('karial',
```

这里的 `dd.items()` 实际上是将 `dd` 转换为可迭代对象，`items()` 方法将字典的元素转化为了元组，而这里 `key` 参数对应的 `lambda` 表达式的意思则是选取元组中的第二个元素作为比较对象，如果写作 `key=lambda item:item[0]` 的话则是选取第一个元素作为比较对象，也就是 `key` 值作为比较对象。

注意排序函数 `sorted()` 返回值是一个 `list`，而原字典中的名值对被转换为了 `list` 中的元组。

0.17 元组

在 Python 中，元组（Tuple）和列表非常相似，与列表不同的是，元组一旦创立，就不可改变，也就是说，元组是不可变的。

0.17.1 创建元组

```
>>> t1 = ()                 # 创建一个空元组
>>> t2 = (11,22,33)         # 创建一个包含三个元素的元组
>>> t3 = tuple([1,2,3,4])   # 使用列表创建元组
>>> t4 = tuple("abc")       # 使用字符串创建元组
```

0.17.2 元组相关方法

元组也是序列，因此序列能使用的方法，如 max , min , len , sum 方法元组也能使用。

```
>>> t1 = (1, 12, 55, 12, 81)
>>> min(t1)
1
>>> max(t1)
81
>>> sum(t1)
161
>>> len(t1)
5
```

0.18 控制声明

在程序中，常常要根据一些条件执行相应的命令。

0.18.1 分支判断

Python 使用 `if-else` 进行控制声明。语法如下：

```
if boolean-expression:
   #statements
else:
   #statements
```

在每一个 if 程序块中，必须使用相同数量的缩进，否则会产生语法错误。这是 Python 和其他语言非常不同的一点。

现在我们看一个例子：

```
i = 11
if i % 2 == 0:
    print("偶数")
else:
    print("奇数")
```

运行结果将根据 i 的值发生变化。

如果需要判断多个条件，我们就可以使用 if-elif-else 控制声明，例如：

```
today = "monday"

if today == "monday":
```

```
    print("this is monday")
elif today == "tuesday":
    print("this is tuesday")
elif today == "wednesday":
    print("this is wednesday")
elif today == "thursday":
    print("this is thursday")
elif today == "friday":
    print("this is friday")
elif today == "saturday":
    print("this is saturday")
elif today == "sunday":
    print("this is sunday")
else:
    print("something else")
```

我们可以根据实际需求，添加对应的多个 elif 条件。

0.18.2 分支嵌套

我们可以在 if 声明语句块中嵌套使用 if 声明。例如：

```
today = "holiday"
bank_balance = 25000
if today == "holiday":
    if bank_balance > 20000:
        print("Go for shopping")
    else:
        print("Watch TV")
else:
    print("normal working day")
```

0.18.3 三元运算符

在其他语言中，有类似 `condition ? true: false` 的三元运算符，在 Python 中，可以这样实现：

```
true if condition else false
```

例如：

```
def b(a):
    return a+2 if a > 10 else 5

print(b(11), b(4))
```

上面的代码将输出 13 6。

0.19 循环

Python 只有两种循环：for 循环和 while 循环。

0.19.1 for 循环

for 循环语法：

```
for i in iterable_object:
   # do something
```

所有在 for 循环或者 while 循环中的声明，必须使用相同的缩进值。否则会出现语法错误。

我们看下面这段代码：

```
mylist = [1, 2, 3, 4]

for i in mylist:
    print(i)
```

在第一次循环时，值 1 被传递给 i，第二次循环时，值 2 被传递给 i。循环一直到列表变量 mylist 没有更多元素时停止。运行结果为：

```
1
2
3
4
```

0.19.2 范围循环

range() 函数能够指定循环的起始值和结束值，从而让循环体在指定的范围内循环。例如：

```
for i in range(10):
    print(i)                    # 0-9
for i in range(1,10):
    print(i)                    # 1-9
for i in range(1,10,2):
    print(i)                    # 1,3,5,7
```

range() 函数只有 1 个参数时，表示从 0 开始循环；两个参数时，第一个参数是起始值，第二个参数是结束值；三个参数时，第三个参数表示循环步长。

0.19.3 while 循环

语法：

```
while condition:
    # do something
```

While 循环会一直执行循环体内部的声明，直到条件变成 false。每次循环都会检查判断条件，如果为真，就继续循环。例如：

```
count = 0

while count < 10:
    print(count)
    count += 1
```

这段代码将会打印出 0-9，直到 count 等于 10。

0.19.4 中断循环

使用 break 语句，可以中断循环，例如：

```
count = 0

while count < 10:
    count += 1
    if count == 5:
        break
    print("inside loop", count)

print("out of while loop")
```

运行结果为：

```
inside loop 1
inside loop 2
inside loop 3
inside loop 4
out of while loop
```

0.19.5 继续循环

当循环体内部出现 continue 声明时，会结束本次循环，跳转到循环体开始位置，开始下一次循环。例如：

```
count = 0

while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print(count)
```

运行结果将打印出 1，3，5，7，9。

0.20 函数

函数是可重用的代码块，使用函数可以帮助我们组织代码的结构。我们创建函数的目的，是能在程序运行中多次使用一系列代码，而不用重复书写代码。

0.20.1 创建函数

Python 使用 def 关键词创建函数，语法如下：

```
def function_name(arg1, arg2, arg3, .... argN):
     #statement inside function
```

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）。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。

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）。有一件事你需要记住：错误的缩进可能会导致错误。

所有在函数内部的声明，都必须使用相等的缩进。函数可以没有参数，也可以有多个参数。多个参数之间用逗号隔开。还可以使用 pass 关键字忽略掉函数主题的声明。

我们看一个函数的例子，下面的函数将计算指定范围的整数之和：

```
def sum(start, end):
    result = 0
    for i in range(start, end + 1):
        result += i
    print(result)

sum(1, 10)
```

在上面的代码中，我们定义了一个叫作 sum() 的函数，该函数有两个参数（start 和 end），该函数将从 start 开始，累加到 end，最后打印出累积之和。代码运行的结果为 55。

0.20.2 函数返回值

上文定义的函数只是简单地在控制台打印出结果，如果我们想要将计算结果赋值给变量，以便做更深入的处理时应该怎么办？当我们遇到这种情况时，可使用 return 语句，将返回函数计算结果并且退出函数。例如：

```
def sumReturn(start, end):
    result = 0
    for i in range(start, end + 1):
        result += i
    return result

a = sumReturn(1, 5)
print(a)
```

在上面这段代码中，我们定义了有返回值的函数 sumReturn()，并将其结果赋值给变量 a。上面代码的运行结果为 15。

当然，return 语句也可以不返回值，而是用来退出函数（实际上会返回 None，为 NoneType 对象）。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。

```
def sum2(start, end):
    if(start > end):
        print("start should be less than end")
        return
    result = 0
    for i in range(start, end + 1):
```

```
        result += i
    return result

s = sum2(110, 50)
print(s, type(s))
```

上述代码的运行结果如下：

```
start should be less than end
None <class 'NoneType'>
```

在 Python 中，如果你不指定 return 的返回值，则会返回 None 值。

0.20.3 全局变量和局域变量

全局变量指的是不属于任何函数，但又可以在函数内外访问的变量。而局域变量指的是在函数内部声明的变量，局域变量只能在函数内部使用，无法在函数外访问（函数执行完后，会销毁内部定义的局部变量）。

下面我们通过例子来演示这两者的区别：

```
global_var = 12          # 定义全局变量

def func():
    local_var = 100      # 定义局部变量
    print(global_var)    # 可以在函数内部访问全局变量

func()                   # 调用函数func()

print(local_var)         # 无法访问变量local_var
```

上述代码将会出现错误：

```
NameError: name 'local_var' is not defined
```

我们再看一个例子：

```
xy = 100          # 定义全局变量xy

def func():
    xy = 200      # 定义局部变量xy
```

```
    print(xy)    # 此时访问的是局部变量xy

func()           # 调用函数func()
```

该代码显示的结果是 200，不是 100。

使用 global 关键字，可以将局部变量同全局变量绑定在一起。例如：

```
t = 1

def increment():
    global t    # 现在的变量t在函数内外都是一致的
    t = t + 1
    print(t)    # 输出 2

increment()
print(t)        # 输出 2
```

使用 global 关键字声明全局变量时，无法直接赋值，比如 “global t = 1” 的写法存在语法错误。

0.20.4 参数的默认值

为参数指定默认值，只需在定义函数时使用赋值语句即可。例如：

```
def func(i, j = 100):
    print(i, j)
```

上述定义的函数 func() 有两个参数 i 和 j。j 的默认值为 100，这意味着我们在调用这个函数的时候可以忽略掉 j 的值，比如 func(2)，运行结果为 2 100。

0.20.5 关键字参数

为函数传递参数值的方法有两种：位置参数和关键字参数。我们之前调用函数的时候都使用的是位置参数。下面我们看如何使用关键字参数：

```
def named_args(name, greeting):
    print(greeting + " " + name)

named_args(name='jim', greeting='Hello')
```

```
named_args(greeting='Hello', name='jim')
named_args('jim', greeting='hello')
```

上述代码运行结果都是 “hello jim”。

关键字参数使用 “name=value” 的名称、值对传递数据，正如上面代码演示的那样，使用关键字参数的时候，参数的顺序是可以调换的，而且位置参数和关键字参数可以混合使用（只能先使用位置参数，后使用关键字参数）。

0.20.6 返回多个值

我们可以通过在 return 语句中使用逗号，将多个值返回，这种返回值的类型是元组。例如：

```
def bigger(a, b):
    if a > b:
        return a, b
    else:
        return b, a

s = bigger(12, 100)
print(s)
print(type(s))
```

运行结果为：

```
(100, 12)
<class 'tuple'>
```

0.20.7 函数文档字符串

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！

```
def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    # 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.
```

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。强烈建议你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__`（注意其中的双下划线）属性（属于函数的名称）来获取函数 print_max 的文档字符串属性。

0.20.8 lambda 表达式

Lambda 表达式（或者 Lambda 形式）用来创建匿名函数。语法如下：

```
lambda_expr        ::=  "lambda" [parameter_list]: expression
```

个人认为，lambda 是为了减少单行函数的定义而存在的，能够提高代码的简洁性。比如：

```
g = lambda x:x+1
```

相当于：

```
def g(x):
    return x+1
```

0.20.9 *args 和 **kwargs

*args 和 **kwargs 是两个魔法变量。那么它们到底是什么？*args 和 **kwargs 主要用于函数定义。你可以将不定数量的参数传递给一个函数。

首先，并不是必须写成 *args 和 **kwargs。只有变量前面的 *（星号）才是必须的. 你也可以写成 *var 和 **vars。而写成 *args 和 **kwargs 只是一个通俗的命名约定。

0.20.9.1 *args 的用法

*args 是用来发送一个非键值对的可变数量的参数列表给一个函数.

这里有个例子帮你理解这个概念:

```
def test_var_args(f_arg, *argv):
    print("first normal arg:", f_arg)
    for arg in argv:
        print("another arg through *argv:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```

这会产生如下输出:

```
first normal arg: yasoob
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: test
```

0.20.9.2 **kwargs 的用法

**kwargs 允许你将不定长度的键值对, 作为参数传递给一个函数。如果你想要在一个函数里处理带名字的参数, 你应该使用 **kwargs。

这里有个让你上手的例子:

```
def greet_me(**kwargs):
    for key, value in kwargs.items():
        print("{0} == {1}".format(key, value))


>>> greet_me(name="yasoob")
name == yasoob
```

现在你可以看出我们怎样在一个函数里，处理了一个键值对参数了。

0.20.9.3 使用 *args 和 **kwargs 来调用函数

那现在我们将看到怎样使用 *args 和 **kwargs 来调用一个函数。假设，你有这样一个小函数：

```
def test_args_kwargs(arg1, arg2, arg3):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("arg3:", arg3)
```

你可以使用 *args 或 **kwargs 来给这个小函数传递参数。下面是怎样做：

```
# 首先使用 *args
>>> args = ("two", 3, 5)
>>> test_args_kwargs(*args)
arg1: two
arg2: 3
arg3: 5

# 现在使用 **kwargs:
>>> kwargs = {"arg3": 3, "arg2": "two", "arg1": 5}
>>> test_args_kwargs(**kwargs)
arg1: 5
arg2: two
arg3: 3
```

0.20.9.4 标准参数与 *args、**kwargs 在使用时的顺序

如果你想在函数里同时使用所有这三种参数，顺序是这样的：

```
some_func(fargs, *args, **kwargs)
```

0.20.10 参考资料

Python 进阶[5]

[5] https://docs.pythontab.com/interpy/

Python 中的面向对象编程

面向对象程序设计（英语：Object-oriented programming，缩写：OOP）是一种程序设计范型，同时也是一种程序开发的方法。对象指的是类的实例。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对电脑下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。

目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。

0.21 Python 对象和类

0.21.1 创建类

Python 一门面向对象的语言。在 Python 中所有的东西都是对象，比如之前学习的整型、字符串等等，甚至模块、函数也都是对象。

面向对象编程时使用对象创建程序，使用对象存储数据和行为。

在 Python 中，使用关键字 class 定义类。类通常包括数据区域，用以存数数据和方法的定义。Python 中的所有类，都包含一个特殊的方法，叫作初始化（initializer），或者叫作构造方法。构造方法会在使用类创建新的对象时自动执行。例如：

```
class Person:

      # 构造函数
      def __init__(self, name):
            self.name = name

      # 定义方法
     def whoami(self):
           return "You are " + self.name
```

上述代码中，我们创建了一个名叫 Person 的类，这个类中包含数据字段 name 和方法 whoami()。

Python 中的所有方法，包括构造方法，首个参数都是 self。这个参数指向对象本身。当我们创建一个新的对象时候，self 参数就会自动指向新创建的对象。

0.21.2 从类中创建对象

使用类名就可创建对象。当我们调用方法时，不需要传递 self 参数，Python 会自动传递。例如：

```
p1 = Person('tom')
print(p1.whoami())
print(p1.name)
```

输出结果为：

```
You are tom
tom
```

我们还可以改变数据字段的值：

```
p1.name = 'jerry'
print(p1.name)
```

输出结果为 jerry。然而，像这样从类的外部获取数据字段，属于不太好的操作方式，下面我们看如何阻止这种操作。

0.21.3 隐藏数据字段

为了隐藏数据字段，我们需要定义私有数据字段。在 Python 中，使用两个前置下划线，就可定义私有数据字段和私有方法。比如：

```
class BankAccount:

    # 构造函数
    def __init__(self, name, money):
        self.__name = name  # 定义私有数据字段
        self.__balance = money  # 定义私有数据字段

    def deposit(self, money):
        self.__balance += money
```

```
    def withdraw(self, money):
        if self.__balance > money:
            self.__balance -= money
            return money
        else:
            return "Insufficient funds"

    def checkbalance(self):
        return self.__balance

b1 = BankAccount('tim', 400)
print(b1.withdraw(500))
b1.deposit(500)
print(b1.checkbalance())
print(b1.withdraw(800))
print(b1.checkbalance())
```

在上述代码中，我们定义了 BankAccout 类，这个类有两个数据字段，但都是私有字段。代码运行结果为：

```
Insufficient funds
900
800
100
```

现在，让我们尝试访问私有数据字段：

```
print(b1.__balance)
```

结果显示：

```
AttributeError: 'BankAccount' object has no attribute '__balance'
```

这就表明，设置为私有的数据字段，无法在类的外部访问。

0.22 操作符重载

我们之前已经看到 + 运算符不但能加数字，还能连接字符串。这之所以可能，是因为 + 运算符在 int 类和 str 类中都被重载。运算符实际上对应着类中相应的方法。为运算符定义方法就是所谓的运算符重载。比如，为让自定义对象能使用 + 运算符，我们需要定义名叫 __add__ 的方法。

让我们看个例子：

```
import math

class Circle:

    def __init__(self, radius):
        self.__radius = radius

    def setRadius(self, radius):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def area(self):
        return math.pi * self.__radius ** 2

    def __add__(self, another_circle):
        return Circle(self.__radius + another_circle.__radius)

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2  # 之所以能使用加法运算符，是因为我们定义了__add__方法
print(c3.getRadius())
```

在上面的例子中，我们为类添加了 ___add___ 方法，该方法允许使用 + 运算符对两个 circle 对象求和。在 ___add___ 方法中，我们创建了一个新的对象，并将其返回给调用者。运行结果如下：

```
4
5
9
```

在 Python 中，除 ___add___ 方法对应 + 运算符之外，还有其他能够重载运算符的方法：如 ___mul___、___sub___ 等等。

0.23 继承和多态

继承（inheritance）允许开发人员先创建一个通用的类，然后扩展为特定类。使用继承机制，我们可以获得类的数据字段和方法，还可以增加自定义的字段和方法，因此，继承提供了一种组织代码、重用代码的方式。

在面向对象的术语中，当类 X 继承自类 Y 时，Y 被叫做超类（super class）或基类（base class），而 X 被成为子类（subclass）或者衍生类（derived class）。

私有数据字段和私有方法只在类的内部使用。子类只能继承父类的非私有数据字段和非私有方法。

继承的语法如下：

```
class SubClass(SuperClass):
  # data fields
  # instance methods
```

让我们看个例子：

```
class Vehicle:

    def __init__(self, name, color):
        self.__name = name      # __name是私有数据字段
        self.__color = color

    def getColor(self):
        return self.__color

    def setColor(self, color):
        self.__color = color

    def getName(self):
        return self.__name

class Car(Vehicle):

    def __init__(self, name, color, model):
        # 调用父类的构造方法
        super().__init__(name, color)
        self.__model = model

    def getDescription(self):
        return self.getName() + self.__model + " in " + self.getColor() + " color"
```

```
c = Car("Ford Mustang", "red", "GT350")
print(c.getDescription())
print(c.getName())
```

上述代码中，我们创建了基类 Vehicle 和子类 Car。在子类 Car 中，我们没有定义 getName() 方法，但我们仍然可以访问 getName()，这是因为类 Car 继承自 Vehicle 类。在这段代码中，super() 方法用来调用基类的方法。上述代码的运行结果如下：

```
Ford MustangGT350 in red color
Ford Mustang
```

0.23.1 多重继承

不像 Java 和 C# 语言，Python 允许多重继承。即一次继承多个基类，比如：

```
class Subclass(SuperClass1, SuperClass2, ...):
   # initializer
   # methods
```

看如下实例：

```
class MySuperClass1():

    def method_super1(self):
        print("method_super1 method called")

class MySuperClass2():

    def method_super2(self):
        print("method_super2 method called")

class ChildClass(MySuperClass1, MySuperClass2):

    def child_method(self):
        print("child method")

c = ChildClass()
c.method_super1()
c.method_super2()
```

输出结果为：

```
method_super1 method called
method_super2 method called
```

因为子类 ChildClass 继承自 MySuperClass1 , MySuperClass2，因此，ChildClass 对象 c 可以访问 method_super1() 方法和 method_super2() 方法。

0.23.2 重写方法

为重写基类的某个方法，子类需要定义一个同名的方法（即拥有相同名称和相同数量的参数）。例如：

```
class A():

    def __init__(self):
        self.__x = 1

    def m1(self):
        print("m1 from A")

class B(A):

    def __init__(self):
        self.__y = 1

    def m1(self):
        print("m1 from B")

c = B()
c.m1()
```

在这段代码中，我们重写了基类的 m1() 方法。输出结果为：

```
m1 from B
```

0.23.3 判断对象是否属于某类

isinstance() 方法用来检测指定对象是否是某个类的实例。例如：

```
>>> isinstance(1, int)
True
```

```
>>> isinstance(1.2, int)
False

>>> isinstance([1,2,3,4], list)
True
```

异常处理

异常是指程序中的例外，违例情况。异常机制是指程序出现错误后，程序的处理方法。当出现错误后，程序的执行流程发生改变，程序的控制权转移到异常处理。{{ “chenjiancun-2015” | cite }}

0.24 捕获异常

异常处理可以使开发人员能以优雅的方式处理错误。

0.24.1 try-except

Python 使用 try-except 语句处理异常。语法如下：

```
try:
    # write some code
    # that might throw exception
except <ExceptionType>:
    # Exception handler, alert the user
```

在 try 语句块中，我们写入可能会产生异常的代码，当异常发生时，try 语句块中的代码会被忽略，转而进入 except 语句块中处理异常。例如：

```
try:
    f = open('somefile.txt', 'r')
    print(f.read())
    f.close()
except IOError:
    print('file not found')
```

上述代码的执行流程如下：

27. 先执行介于 try 和 except 之间的语句。
28. 如果没有异常，则 except 语句块中的代码会被跳过。

29. 如果文件不存在，则产生异常，在 try 语句块中的其他代码会被跳过。
30. 当异常发生时，如果异常类型和 except 关键字后的异常名称匹配，就执行 except 分支中的代码。上述代码中只能处理 IOError 异常，如要处理其他类型的异常，还需要添加更多的 except 分支。

0.24.2 多个 except

try 声明可以有多个 except 分支，它还可以选择 else 和 finally 分支。语法如下：

```
try:
    <body>
except <ExceptionType1>:
    <handler1>
except <ExceptionTypeN>:
    <handlerN>
except:
    <handlerExcept>
else:
    <process_else>
finally:
    <process_finally>
```

except 分支类似于 elif。当异常发生时，将检查 except 分支是否和异常类型匹配。如果匹配，就执行对应 except 分支中的代码。在最后一个 except 分支中，异常类型是被忽略了的。如果异常发生，但没有匹配到最后一个 except 之前的分支，则最后的 except 分支中的代码会被执行。如果没有任何异常发生，则执行 else 语句中的代码。finally 分支中的代码，无论是否有异常发生，都会被执行。例如：

```
try:
    num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
    result = num1 / num2
    print("Result is", result)

except ZeroDivisionError:
    print("Division by zero is error !!")

except SyntaxError:
    print("Comma is missing. Enter numbers separated by comma like this 1, 2")

except:
    print("Wrong input")
```

```
else:
    print("No exceptions")

finally:
    print("This will execute no matter what")
```

eval() 函数允许在 Python 程序内部运行 Python 代码，了解更多关于 eval() 的信息，请访问http://stackoverflow.com/questions/9383740/what-does-pythons-eval-do

0.24.3 自定义异常

使用关键字 raise，可以在方法中自定义异常。语法为：

```
raise ExceptionClass("Your argument")
```

例如：

```
def enterage(age):
    if age < 0:
        raise ValueError("Only positive integers are allowed")

    if age % 2 == 0:
        print("age is even")
    else:
        print("age is odd")

try:
    num = int(input("Enter your age: "))
    enterage(num)

except ValueError:
    print("Only positive integers are allowed")
except:
    print("something is wrong")
```

当用户输入的年龄小于 0 时，程序显示结果为：

```
only positive integers are allowed
```

Python 装饰器

装饰器本质上是一个 Python 函数或类，它可以让其他函数或类在不需要做任何代码修改的前提下增加额外功能，装饰器的返回值也是一个函数/类对象。有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码到装饰器中并继续重用。概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

0.25 为什么要用装饰器

先来看一个简单例子，虽然实际代码可能比这复杂很多，假如已经有如下函数，用来输出特定信息，并且这个函数已被其他程序片段使用多次：

```
def foo():
    print('i am foo')
```

现在有一个新的需求，希望可以记录下函数的执行日志，于是在代码中添加日志代码：

```
def foo():
    print('i am foo')
    logging.info("foo is running")
```

如果函数 bar()、bar2() 也有类似的需求，怎么做？再写一个 logging 在 bar 函数里？这样就造成大量雷同的代码，为了减少重复写代码，我们可以这样做，重新定义一个新的函数，专门处理日志，日志处理完之后再执行真正的业务代码：

```
def use_logging(func):
    logging.warn("%s is running" % func.__name__)
    func()

def foo():
    print('i am foo')

use_logging(foo)
```

这样做逻辑上是没问题的，功能是实现了，但是我们调用的时候不再是调用真正的业务逻辑 foo 函数，而是换成了 use_logging 函数，这就破坏了原有的代码结构，现在我们不得不每次都要把原来的那个 foo 函数作为参数传递给 use_logging 函数，那么有没有更好的方式的呢? 当然有，答案就是装饰器。

0.26 简单装饰器

```
def use_logging(func):

    def wrapper():
        logging.warn("%s is running" % func.__name__)
        return func()   # 把 foo 当做参数传递进来时，执行func()就相当于执行foo()
    return wrapper

def foo():
    print('i am foo')

foo = use_logging(foo)  # 因为装饰器 use_logging(foo) 返回的时函数对象 wrapper，这条语句相当于
foo()                   # 执行foo()就相当于执行 wrapper()
```

use_logging() 就是一个装饰器，它一个普通的函数，它把执行真正业务逻辑的函数 func 包裹在其中，看起来像 foo 被 use_logging 装饰了一样，use_logging 返回的也是一个函数，这个函数的名字叫 wrapper。在这个例子中，函数进入和退出时，被称为一个横切面，这种编程方式被称为面向切面的编程。

0.27 @ 语法糖

如果你接触 Python 有一段时间了的话，想必你对 @ 符号一定不陌生了，没错 @ 符号就是装饰器的语法糖，它放在函数开始定义的地方，这样就可以省略最后一步再次赋值的操作。

```
def use_logging(func):

    def wrapper():
        logging.warn("%s is running" % func.__name__)
        return func()
    return wrapper

@use_logging
def foo():
```

```
    print("i am foo")

foo()
```

如上所示，有了 @ ，我们就可以省去 foo = use_logging(foo) 这一句了，直接调用 foo() 即可得到想要的结果。你们看到了没有，foo() 函数不需要做任何修改，只需在定义的地方加上装饰器，调用的时候还是和以前一样，如果我们有其他的类似函数，我们可以继续调用装饰器来修饰函数，而不用重复修改函数或者增加新的封装。这样，我们就提高了程序的可重复利用性，并增加了程序的可读性。

装饰器在 Python 使用如此方便都要归因于 Python 的函数能像普通的对象一样能作为参数传递给其他函数，可以被赋值给其他变量，可以作为返回值，可以被定义在另外一个函数内。

0.28 *args、**kwargs

可能有人问，如果我的业务逻辑函数 foo 需要参数怎么办？比如：

```
def foo(name):
    print("i am %s" % name)
```

我们可以在定义 wrapper 函数的时候指定参数：

```
def wrapper(name):
        logging.warn("%s is running" % func.__name__)
        return func(name)
    return wrapper
```

这样 foo 函数定义的参数就可以定义在 wrapper 函数中。这时，又有人要问了，如果 foo 函数接收两个参数呢？三个参数呢？更有甚者，我可能传很多个。当装饰器不知道 foo 到底有多少个参数时，我们可以用 *args 来代替：

```
def wrapper(*args):
        logging.warn("%s is running" % func.__name__)
        return func(*args)
    return wrapper
```

如此一来，甭管 foo 定义了多少个参数，我都可以完整地传递到 func 中去。这样就不影响 foo 的业务逻辑了。这时还有读者会问，如果 foo 函数还定义了一些关键字参数呢？比如：

```
def foo(name, age=None, height=None):
    print("I am %s, age %s, height %s" % (name, age, height))
```

这时，你就可以把 wrapper 函数指定关键字函数：

```
def wrapper(*args, **kwargs):
        # args是一个数组，kwargs一个字典
        logging.warn("%s is running" % func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

0.29 带参数的装饰器

装饰器还有更大的灵活性，例如带参数的装饰器，在上面的装饰器调用中，该装饰器接收唯一的参数就是执行业务的函数 foo 。装饰器的语法允许我们在调用时，提供其它参数，比如 @decorator(a)。这样，就为装饰器的编写和使用提供了更大的灵活性。比如，我们可以在装饰器中指定日志的等级，因为不同业务函数可能需要的日志级别是不一样的。

```
def use_logging(level):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if level == "warn":
                logging.warn("%s is running" % func.__name__)
            elif level == "info":
                logging.info("%s is running" % func.__name__)
            return func(*args)
        return wrapper

    return decorator

@use_logging(level="warn")
def foo(name='foo'):
    print("i am %s" % name)

foo()
```

上面的 use_logging 是允许带参数的装饰器。它实际上是对原有装饰器的一个函数封装，并返回一个装饰器。我们可以将它理解为一个含有参数的闭包。当我们使用 @use_logging(level="warn") 调用的时候，Python 能够发现这一层的封装，并把参数传递到装饰器的环境中。

@use_logging(level="warn") 等价于 @decorator

0.30 类装饰器

没错，装饰器不仅可以是函数，还可以是类，相比函数装饰器，类装饰器具有灵活度大、高内聚、封装性等优点。使用类装饰器主要依靠类的 `__call__` 方法，当使用 `@` 形式将装饰器附加到函数上时，就会调用此方法。

```
class Foo(object):
    def __init__(self, func):
        self._func = func

    def __call__(self):
        print ('class decorator runing')
        self._func()
        print ('class decorator ending')

@Foo
def bar():
    print ('bar')

bar()
```

0.31 functools.wraps

使用装饰器极大地复用了代码，但是他有一个缺点就是原函数的元信息不见了，比如函数的 `docstring`、`__name__`、参数列表，先看例子：

```
#装饰器
def logged(func):
    def with_logging(*args, **kwargs):
        print func.__name__      # 输出 'with_logging'
        print func.__doc__       # 输出 None
        return func(*args, **kwargs)
    return with_logging

# 函数
@logged
def f(x):
   """does some math"""
   return x + x * x

logged(f)
```

不难发现，函数 f 被 with_logging 取代了，当然它的 docstring，__name__ 就是变成了 with_logging 函数的信息了。好在我们有 functools.wraps，wraps 本身也是一个装饰器，它能把原函数的元信息拷贝到装饰器里面的 func 函数中，这使得装饰器里面的 func 函数也有和原函数 foo 一样的元信息了。

```
from functools import wraps
def logged(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print func.__name__      # 输出 'f'
        print func.__doc__       # 输出 'does some math'
        return func(*args, **kwargs)
    return with_logging

@logged
def f(x):
   """does some math"""
   return x + x * x
```

0.32 装饰器顺序

一个函数还可以同时定义多个装饰器，比如：

```
@a
@b
@c
def f ():
    pass
```

它的执行顺序是从里到外，最先调用最里层的装饰器，最后调用最外层的装饰器，它等效于

```
f = a(b(c(f)))
```

参考资料

1. 理解 Python 装饰器[6]

[6] https://foofish.net/python-decorator.html

模块

Python 模块是一个包含有函数、变量、类和常量等等内容的 python 文件。模块帮助我们将相关的代码组织在一起，例如 `math` 模块拥有数学相关的函数。

0.33 创建模块

创建一个名为 `mymodule.py` 的新文件，并写入下面的代码：

```
foo = 100

def hello():
    print("i am from mymodule.py")
```

在这个文件中，我们定义了一个全部变量 `foo` 和一个名为 `hello()` 的方法。现在我们可以使用 `import` 关键词来引入这个模块，并使用 `mymodule.py` 中的变量和函数：

```
import mymodule

print(mymodule.foo)
mymodule.hello()
```

上述代码的运行结果如下：

```
100
i am from mymodule.py
```

如之前代码所示，调用模块的变量和函数时，需要指定模块的名称。

0.34 使用模块中的指定内容

当我们使用 `import` 声明导入模块时，模块中的所有内容都被导入到当前文件中。如果我们只需要模块中的个别内容时该如何操作呢？使用 `from` 关键词，就可以达到这样的目的，比如：

```
from mymodule import foo
print(foo)
```

上述代码的运行结果为 100。

当使用 `from improt` 语句导入特定内容后，访问这些内容就不需要再使用模块名了。

0.35 dir 函数

内置的 `dir()` 函数能够返回由对象所定义的名称列表。如果这一对象是一个模块，则该列表会包括函数内所定义的函数、类与变量。该函数接受参数。如果参数是模块名称，函数将返回这一指定模块的名称列表。如果没有提供参数，函数将返回当前模块的名称列表。

```
>>> import sys

# 给出 sys 模块中的属性名称
>>> dir(sys)
['__displayhook__', '__doc__',
'argv', 'builtin_module_names',
'version', 'version_info']
# only few entries shown here

# 给出当前模块的属性名称
>>> dir()
['__builtins__', '__doc__',
'__name__', '__package__']

# 创建一个新的变量 'a'
>>> a = 5

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a']
```

0.36 包

包是指一个包含模块与一个特殊的 `__init__.py` 文件的文件夹，后者向 Python 表明这一文件夹是特别的，因为其包含了 Python 模块。

假设你想创建一个名为 “world” 的包，其中还包含着”asia“、”africa“等其它子包，同时这些子包都包含了诸如”india“、”madagascar“等模块。下面是你会构建出的文件夹的结构：

```
- <some folder present in the sys.path>/
    - world/
        - __init__.py
        - asia/
            - __init__.py
            - india/
                - __init__.py
                - foo.py
        - africa/
            - __init__.py
            - madagascar/
                - __init__.py
                - bar.py
```

包是一种能够方便地分层组织模块的方式。

第二部分

进阶

内置函数

所谓内置函数，就是 Python 解释器已经拥有的一系列函数和类型，这些函数一直可用，无需定义。

0.37 dict 函数

0.37.1 描述

dict() 函数用于创建一个字典。

0.37.2 语法

```
class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)
```

参数说明：

```
**kwargs -- 关键字
mapping -- 元素的容器。
iterable -- 可迭代对象。
```

0.37.3 返回值

返回一个字典。

0.37.4 实例

以下实例展示了 dict 的使用方法：

```
>>>dict()                          # 创建空字典
{}
>>> dict(a='a', b='b', t='t')      # 传入关键字
{'a': 'a', 'b': 'b', 't': 't'}
>>> dict(zip(['one', 'two', 'three'], [1, 2, 3]))   # 映射函数方式来构造字典
{'three': 3, 'two': 2, 'one': 1}
>>> dict([('one', 1), ('two', 2), ('three', 3)])    # 可迭代对象方式来构造字典
{'three': 3, 'two': 2, 'one': 1}
>>>
```

0.38 zip 函数

0.38.1 描述

将不同迭代对象中的元素整合为一个迭代对象。

0.38.2 语法

```
zip(*iterables)
```

0.38.3 返回值

返回元组。

0.38.4 特殊用法

zip() 方法和 * 运算符连用时，用来拆解一个列表。

0.38.5 实例

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
```

```
>>> x == list(x2) and y == list(y2)
True
```

0.39 list 函数

0.39.1 描述

与其说 list() 是函数，不如说它是一个可变序列的数据类型，其作用是将数据转化为列表。

0.39.2 语法

```
class list([iterable])
```

0.39.3 返回值

可变序列

0.39.4 实例

```
>>> list('world')
['w', 'o', 'r', 'l', 'd']
```

0.40 min 函数

0.40.1 描述

求多个参数中的最小值，或者是可迭代数据中的最小元素。

0.40.2 语法

```
min(iterable, *[, key, default])
min(arg1, arg2, *args[, key])
```

0.40.3 返回值

最小的元素，可能是字符串、数字，也可能是元组、列表。

0.40.4 实例

```
>>> min(1,2,3)  # 求三个元素中的最小值
1
>>> min(1,'2') # 不同类型的变量无法直接求最小值
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
>>> min('2','3') # 可以对字符串求最小值，按字母顺序求值
'2'
>>> min(-1,-2) # 可以对负数求最小值
-2
>>> min(-1,-2,key = abs) # key参数可以是函数，例如abs()
-1
>>> min(-1,'-2',key = int) # key参数为类型转换函数
'-2'
>>> min([1,2],(1,1)) # 无法直接对元素和列表求最小值
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'tuple' and 'list'
>>> min([1,2],(1,1),key = lambda x:x[1]) # key值可以是列表中的某个元素
(1, 1)
>>> min([1,2],(1,3),key = lambda x:x[1])
[1, 2]
>>> min([1,2,3],(1,3,3),key = lambda x:x[1])
[1, 2, 3]
>>> min([1,2,3],(1,3,3),key = lambda x:x[2])
[1, 2, 3]
>>> min([1,4,3],(1,3,3),key = lambda x:x[2])
[1, 4, 3]
>>> min([1,4,3],(1,3,3),key = lambda x:x[0])
[1, 4, 3]
>>> min([1,4,3],(1,3,3),key = lambda x:x[1])
(1, 3, 3)
>>> min([1,4,3],(1,3,3),key = lambda x:x[1])
(1, 3, 3)
```

Python 可迭代对象

迭代器（iterator）有时又称游标（cursor）是程式设计的软件设计模式，可在容器物件（container，例如链表或阵列）上遍访的界面，设计人员无需关心容器物件的内存分配的实现细节。

0.41 可迭代对象

Python 中经常使用 for 来对某个对象进行遍历，此时被遍历的这个对象就是可迭代对象，像常见的序列（字符串、列表、元组）、字典都是。如果给一个准确的定义的话，就是只要它定义了可以返回一个迭代器的 __iter__ 方法，或者定义了可以支持下标索引的 __getitem__ 方法，那么它就是一个可迭代对象。

0.42 迭代器

迭代器是通过 next() 来实现的，每调用一次他就会返回下一个元素，当没有下一个元素的时候返回一个 StopIteration 异常，所以实际上定义了这个方法的都算是迭代器。

0.43 生成器

生成器是构造迭代器的最简单有力的工具，与普通函数不同的只有在返回一个值的时候使用 yield 来替代 return，然后 yield 会自动构建好 next() 和 iter()。

0.44 三者之间关系

1. 可迭代对象包含迭代器。
2. 如果一个对象拥有 __iter__ 方法，其是可迭代对象；如果一个对象拥有 next 方法，其是迭代器。
3. 定义可迭代对象，必须实现 __iter__ 方法；定义迭代器，必须实现 __iter__ 和 next 方法。

0.45 迭代器长度的计算

迭代器（包括生成器）是不能直接使用 len() 方法计算长度的，例如：

```
>>>l = (i for i in xrange(100) if i&1)

>>>len(l)

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: object of type 'generator' has no len()
```

计算迭代器的长度，我们可以先将其转化为列表再计算，但如果迭代器规模较大，这将消耗大量内存，并不是很好的解决方案：

```
>>>len(list(l))
```

我们可以使用更为简洁的方式，即通过循环求和的方式得到迭代器的长度：

```
>>sum(1 for _ in l)
50
```

在此基础上，我们可以定义一个函数，专门用来求迭代器（生成器）的长度：

```
def leniter(iterator):
    """leniter(iterator): return the length of an iterator,consuming it."""
    if hasattr(iterator, "__len__"):
        return len(iterator)
    nelements = 0
    for _ in iterator:
        nelements += 1
    return nelements
```

0.46 yield

可以将 yield 简单类比为 return，但是它除了返回一个值，还会记住这个返回的位置，下次迭代就从这个位置后 (下一行) 开始。yield 与 retrun 语句最大的差别是，return 语句之后的代码是不执行的，而 yield 语句之后的代码依然得到执行。

```
def yield_test(n):
    for i in range(n):
```

```
        yield call(i)
        print("i=",i)
        print("yield函数第 %d 次迭代结束"%(i))
    print('yield函数整体结束')

def call(i):
    return i*2

#使用for循环获取迭代器中的值
for i in yield_test(5):
    print('yield 函数的返回值是:',i)
```

上述代码的输出结果为:

```
yield 函数的返回值是: 0
i= 0
yield函数第 0 次迭代结束
yield 函数的返回值是: 2
i= 1
yield函数第 1 次迭代结束
yield 函数的返回值是: 4
i= 2
yield函数第 2 次迭代结束
yield 函数的返回值是: 6
i= 3
yield函数第 3 次迭代结束
yield 函数的返回值是: 8
i= 4
yield函数第 4 次迭代结束
yield函数整体结束
```

一个带有 yield 的函数就是一个 generator，它和普通函数不同，生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 `next()`（在 `for` 循环中会自动调用 `next()`）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 `next()` 的值，不仅代码简洁，而且执行流程异常清晰。

单、双下划线的区别

在学习 Python 的时候，估计会对 Python 代码中的下划线产生困惑，现总结如下：

0.47 单下划线开头

在 Python 中不存在真正意义上的私有方法或者属性，前面加单下划线 _ 只是表示你不应该去访问这个方法或者属性，因为它不是 API 的一部分。

```
class BaseForm(StrAndUnicode):
    ...

    def _get_errors(self):
        "Returns an ErrorDict for the data provided for the form"
        if self._errors is None:
            self.full_clean()
        return self._errors

    errors = property(_get_errors)
```

这段代码的设计就是 errors 属性是对外 API 的一部分，如果你想获取错误详情，应该访问 errors 属性，而不是（也不应该）访问 _get_errors 方法。

0.48 双下划线开头

设计双下划线开头的初衷和目的，是为了避免子类覆盖父类的方法。

```
class A(object):

    def __method(self):
        print("I'm a method in class A")

    def method_x(self):
```

```
        print("I'm another method in class A\n")

    def method(self):
        self.__method()
        self.method_x()


class B(A):

    def __method(self):
        print("I'm a method in class B")

    def method_x(self):
        print("I'm another method in class B\n")


if __name__ == '__main__':

    print("situation 1:")
    a = A()
    a.method()

    b = B()
    b.method()

    print("situation 2:")
    a._A__method()
```

执行结果：

```
situation 1:
I'm a method in class A
I'm another method in class A

I'm a method in class A
I'm another method in class B

situation 2:
I'm a method in class A
```

0.49 双下划线开头和结尾

一般来说像 `__this__` 这种开头结尾都加双下划线的方法表示这是 Python 自己调用的，你不要调用。比如我们可以调用 `len()` 函数来求长度，其实它后台是调用了 `__len__()` 方法。一般我们应该使用 `len`，而不是直接使用 `__len__()`。正如下面的例子：

```
class Room(object):

    def __init__(self):
        self.people = []

    def add(self, person):
        self.people.append(person)

    def __len__(self):
        return len(self.people)

room = Room()
room.add("Igor")
print len(room)
```

这个例子中，因为我们实现了 `__len__()`，所以 `Room` 对象也可以使用 `len` 函数了。

正则表达式

信息技术日新月异，有些已成昨日黄花，然而伟大的东西却弥久如新，正则表达式 (Regular Expression) 就是这样一种伟大的发明。它是一个强大、便捷、高效的文本处理工具，能成百倍地提高开发效率和程序质量。正则表达式绝对值得每一个程序员掌握，甚至值得所有知识工作者了解。

然而奇怪的是，这样一个了不起的技术，在我国却没有得到充分推广。究其原因，主要有二：其一，正则表达式产生和发展与 UNIX 文化体系中，而我国的信息技术教学长期受到微软文化的影响。其二，正则表达式并不是那么容易掌握，需要正确的方法。学习正则表达式，入门不难，看一些例子，试着模仿，就可粗通。但要真正掌握，还需了解正则表达式的原理，强烈建议阅读 Jeffrey Firedl 的《精通正则表达式》。这本书自 1997 年出版后，凭借其质量，成为正则表达式图书领域当之无愧的“圣经”。

0.50 正则表达式发展简史

关于正则表达式，最初的想法来自 20 世纪 40 年代的两位神经学家，他们研究出一种模型，描述神经系统在神经元层面上的工作模式。若干年后，数学家 Stephen Kleene 在代数学中正式描述了这种模型，并将它命名为正则集合。Stephen 发明了一套简洁的表明正则集合的方法，他称之为“正则表达式”。

20 世纪 50 年代和 60 年代，理论数学界对正则表达式进行了充分的研究。20 世纪 70 年代，开始将正则表达式应用到计算方面，再到后来正则表达式开始应用到其它领域 (文本编辑、生物信息、基因图谱等等)。

在 UNIX 中 ed 编辑器开始使用正则表达式，其中的一个功能最终发展成独立的工具 grep(Global Regular Expression Print)。这个工具在众多程序员的改进下，功能日渐强大。1986 年，Henry Spencer 发布了用 C 语言写的正则表达式包，这是一个具有开创性意义的包，因为它可以毫无困难地移植到其他程序中。

1987 年，Larry Well 发布了 Perl 语言的第一个版本，提供了正则表达式操作符，在脚本语言中是首创。之后，Perl 语言越来越强壮，错误也越来越少，添加了许多新的正则表达式特性。

其他语言的开发人员借鉴了 Perl 语言的正则表达式特性，最终在某种程度上“兼容 Perl”，如 Python、Ruby、Java、PHP、C/C++、.NET 都有各自的正则表达式包。1997 年，Philip Hazel 开发了 PCRE，这是一套兼容 Perl 正则表达式的库，PCRE 的正则引擎质量很高，全面仿制 Perl 的正则表达式语法和语义。

0.51 Python 中使用正则表达式的流程

正则表达式的大致匹配过程是：依次拿出表达式和文本中的字符比较，如果每一个字符都能匹配，则匹配成功；一旦有匹配不成功的字符则匹配失败。如果表达式中有量词或边界，这个过程会稍微有一些不同，但也大致相同。

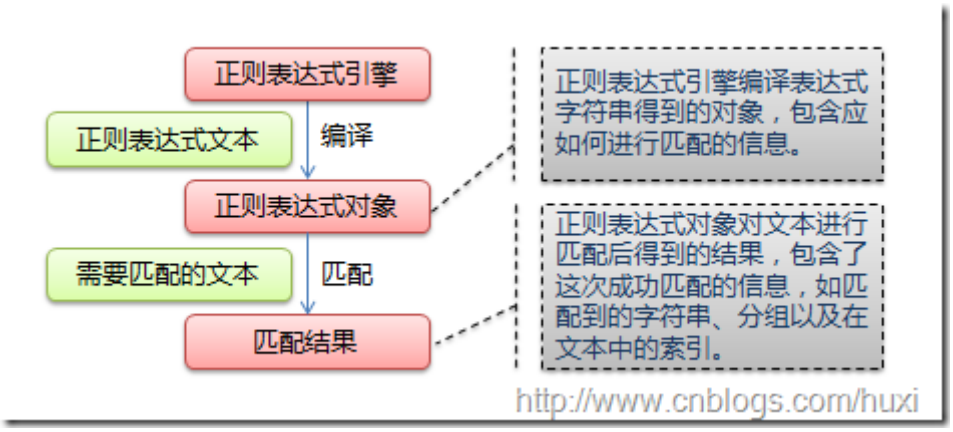

图 1: 正则表达式

0.52 正则表达式的构成要素

正则表达式由四部分组成: 定界符、原子、元字符、模式修饰符。

在 python 中，定界符可以不用写，使用 raw 字符串来表示正则表达式，更加方便。例如 `r'\d?i'` 就表示是一个正则表达式，包含了元字符和模式修饰符。

0.52.1 定界符

定界符可以是任意非字母数字、非反斜线、非空白字符，常用“/”。

放在定界符中的最小的一个匹配单位，在一个正则表达式中，至少要有一个原子。正则表达式是一个从左到右匹配目标字符串的模式。

正则表达式的威力源于它可以在模式中拥有选择和重复的能力。一些字符被赋予特殊的涵义，使其不再单纯的代表自己，模式中的这种有特殊涵义的编码字符称为元字符。元字符用来修饰原子。

模式修饰符用来修正正则表达式进行，放在最右边。

0.52.2 原子

正则表达式中的原子可以是打印字符 (键盘上的各种字母, 有的需要转义)、非打印字符 (如回车、制表符等)、转义符、子表达式 (方括弧、圆括弧)。我们在理解正则表达式的时候，应该按

照原子的顺序来解读。其他的子表达式 (虽然是多个字符构成，但在逻辑上是一个原子)，将在后续的内容进行详细讲解。

0.53 元字符

模式中有特殊涵义的编码字符称为元字符。元字符被赋予特殊的涵义，使其不再单纯的代表自己。如:

\d 表示数字，并不表示字母 d。

0.53.1 匹配单个字符的元字符

可以使用元字符匹配指定的单个字符，这是正则表达式的基础，正则表达式的其它功能都建立在对单个字符的匹配之上。

符号	元字符	匹配对象
.	点号	匹配单个任意字符（除换行）
[……]	字符组	匹配单个列出的字符，可以使用连字符 “-” 表示范围
[\^……]	排除型字符组	匹配单个未列出的字符
\char	转义字符	如 char 是元字符，匹配 char 对应的普通字符

字符组（排除型字符组）是子表达式的一种，用方括弧来表示。方括弧 [] 内的多个字符会被认为是可选的一个字符。例如 [123456] 匹配 1 到 6 中的任意一个数字。[123456] 还可以写成 [1-6]。

0.53.1.1 正则表达式中的特殊字符

正则表达式特殊字符有:

```
. \ + * ? [ ^ ] $ ( ) { } = ! < > | : -
```

当这些字符及定界符需要在模式中进行匹配时，需要在其前面加上反斜线进行转义。

0.53.2 常用转义符

常用转义符见下表:

符号	含义
\t	制表符
\n	换行符

符号	含义
\r	回车符
\s	任何“空白”字符（例如空格符、制表符、进纸符、回车换行符等）
\S	除\s 之外的任何字符
\w	[a-zA-Z0-9]
\W	除\w 之外的任何字符
\d	数字
\D	除\d 外的任何字符

如果要在正则表达式中使用反斜线，需要四个连用\\\\ ，这里为什么是四个反斜线，原因在于：我们知道在正则概念上转义反斜线的写法为：\\ 这个在正则表式下是能匹配出\，这是正则表达式引擎拿到的模式，但你也注意到了匹配出的\ 会转义后面的分隔符，所以我还需要一个反斜线来转义这个\ 。这样四个反斜线可理解为: 前两个“生成”的\ 转义后两个“生成”的反斜杠。

0.53.3 提供计数功能的元字符

元字符还可以用来表示前置字符的数量，尤其是我们不知道字符内容的时候，这个功能就非常有用和高效。

符号	元字符	匹配对象
?	问号	匹配前置字符 0-1 次
*	星号	匹配前置字符 0-N 次
+	加号	匹配前置字符 1-N 次
{n,m}	区间量词	匹配前置字符 n-m 次

默认情况下，量词都是“贪婪”的，也就是说，它们会在不导致模式匹配失败的前提下，尽可能多的匹配字符 (直到最大允许的匹配次数)。然而，如果一个量词紧跟着一个 ?(问号) 标记，它就会成为懒惰 (非贪婪) 模式，它不再尽可能多的匹配，而是尽可能少的匹配。

0.53.4 匹配位置的元字符

元字符还可以用来表示位置，如一行的开始，单词的边界，或者是指定的某个位置。这种表示位置的元字符，又叫断言。一个断言就是一个对当前匹配位置之前或之后的字符的测试，它不会实际消耗任何字符。

符号	元字符	匹配对象
^	脱字符	匹配一行的开头位置
$	美元符	匹配一行的结束位置
\b	单词分界符	单词的分界位置

符号	元字符	匹配对象
\B	单词分界符	除单词分界位置的任何位置
(?<=...)	环视	匹配指定的位置

注意在子表达式内部，脱字符 ^ 并不表示开头，而是表示“非”。

复杂的断言以子组的方式编码。它有两种类型：前瞻断言 (匹配到的字符组的前面) 和后瞻断言 (匹配到的字符组的后面)，每个类型中又分为肯定形式和否定形式。

0.53.4.1 前瞻断言

前瞻断言的肯定形式为 (?=……)。

```
$pattern = '/(?=中华人民共和国)/';
// (?=中华人民共和国)匹配中华人民共和国的开始位置
$string  = '中华人民共和国，中华人民共和国，中华民国';
echo preg_replace($pattern, '我爱', $string), "<br />";
```

运行结果为：

```
我爱中华人民共和国，我爱中华人民共和国，中华民国
```

前瞻断言的否定形式为 (?!……)：

```
$pattern = '/(?!中华人民共和国)中华/';
// // (?!中华人民共和国)匹配不是中华人民共和国的开始位置
$string  = '中华人民共和国，中华人民共和国，中华民国';
echo preg_replace($pattern, '我爱中华', $string), "<br />";
```

运行结果为：

```
中华人民共和国，中华人民共和国，我爱中华民国
```

0.53.4.2 后瞻断言

后瞻断言的肯定形式为 (?<=……)：

```
$pattern = '/(?<=中华人民共和国)/';
// (?<=中华人民共和国)匹配中华人民共和国的结束位置
```

```
$string  = '中华人民共和国，中华人民共和国，中华民国';
echo preg_replace($pattern, '威武', $string), "<br />";
```

运行结果为：

```
中华人民共和国威武，中华人民共和国威武，中华民国
```

后瞻断言的否定形式为：(?<!……)

```
$pattern = '/国(?<!中华人民共和国)/';
// // (?<!中华人民共和国)匹配不是中华人民共和国的结束位置
$string  = '中华人民共和国，中华人民共和国，中华民国，美国';
echo preg_replace($pattern, '国人民', $string), "<br />";
```

运行结果为：

```
中华人民共和国，中华人民共和国，中华民国人民，美国人民
```

0.53.4.3 断言的组合

多个断言可以同时出现。断言子表达式的顺序无前后区分，只要同时满足断言条件的位置，都符合正则表达式模式要求。比如：

```
(?<=\d{3})(?<!999)foo
```

匹配前面有三个数字但不是“999”的字符串“foo”。

0.53.5 其他元字符

还有一些元字符用来分割表达式、设定引用方式等。

符号	元字符	匹配对象
|	选择符	匹配任意分割的表达式
(……)	括号	限定多选结构的范围，标注量词的作用范围，为反向引用捕获文本
/1,/2,……	反向引用	匹配之前的括号内的子表达式匹配的文本

竖线字符用于分离模式中的可选路径。比如模式 com|cn 匹配“com”或者“cn”。竖线可以在模式中出现任意多个，并且允许有空的可选路径（匹配空字符串）。匹配的处理从左到右尝试每一个可选路径，并且使用第一个成功匹配的。

括号中匹配的内容，可以保存到变量中，命名的规则为 (?<name>)：

```
$pattern = '/<b>(?<title>.*?)<\/b>/s';
```

0.54 模式修饰符

模式修饰符是对整个正则表达式功能的调整。常用的模式修饰符如下：

1. i 表示匹配大小写，在 re 模块中，可以使用 re.I 表示；
2. m 表示匹配多行，在 re 模块中，可以使用 re.M 表示；
3. s 可以点号元字符匹配所有字符，包含换行，在 re 模块中，可以使用 re.S 表示。

0.55 在 Python 中使用正则表达式

Python 的 re 库全面实现了正则表达式的功能，下面将就常用功能进行示例。

0.55.1 search()

search() 方法会扫描整个字符串，然后返回第一个成功匹配的结果，如果没有，则返回 None。

0.55.2 compile()

compile() 方法可以将正则字符串编译成正则表达式对象，以达到复用的目的，也还可以传入模式修饰符。

0.55.3 findall()

findall() 方法会搜索整个字符串，然后返回所有匹配结果，如果没有，则返回 None。

```
import re

content = 'dfdsfDDDDZ 23234'
pattern = re.compile('[a-z]', re.I)
result = re.search(pattern, content)
print(result)
print(result.group())
print(result.span())
result = re.findall(pattern, content)
print(result)
```

上面的代码运行结果如下：

```
<re.Match object; span=(0, 1), match='d'>
d
(0, 1)
['d', 'f', 'd', 's', 'f', 'D', 'D', 'D', 'D', 'Z']
```

可以看出，search() 方法和 findall() 方法存在的区别。

命令行参数模块 argparse

Argparse 模块主要用来开发类似于 shell 中原生命令那样用户友好的命令行工具。使用该模块可以定义必需参数、可选参数，还能自动生成帮助和使用说明。

先看一个简单例子：

```
#! /Users/ncsxbmu/anaconda3/bin/python
# coding=utf-8

import sys
print ("文件名 = ", sys.argv[0])
for i in range(1, len(sys.argv)):
    print ("参数%s = %s"%(i, sys.argv[i]))
```

假设上述代码存放在名为 test.py 的文件中，则上述代码输出结果如下：

```
 # 不带参数调用
   python test.py
file =  test.py

# 带多个参数调用
   python test.py 1 3
file =  test.py
参数1 = 1
参数2 = 3
```

从这个例子中我们可以看出，利用内置模块 sys.argv 能非常方便地获取参数内容。但这个模块在处理复杂参数时不够简洁和方便。因此，我们需要更加强大的 argparse 模块，该模块的用法是：

1. 创建解析器
2. 添加参数
3. 解析参数

下面分别讲解：

0.56 创建解析器

使用 `ArgumentParser` 类创建参数解析器，参数都为关键字参数。语法为：

```
class argparse.ArgumentParser(prog=None, usage=None, description=None, epilog=None, parents=[],
```

其中主要参数说明如下：

1. prog - 程序名称，默认值为程序文件名。
2. usage - 程序用法描述，默认根据添加的参数生成。
3. description - 参数说明信息之前的文本默认为空。
4. epilog - 参数说明信息之后的文本，默认为空。
5. parents - 需要包含的父解析器。
6. add_help - 添加 -h/–help 选项，默认为真。
7. allow_abbrev - 是否允许参数缩写，默认为真。

例如：

```
#! /Users/ncsxbmu/anaconda3/bin/python
# coding=utf-8

import argparse

parser = argparse.ArgumentParser()
parser = argparse.ArgumentParser(description = '合并多个markdown文件并转化为docx文件')
parser.print_help()
```

运行结果如下：

```
    python test.py

usage: test.py [-h]

合并多个markdown文件并转化为docx文件

optional arguments:
  -h, --help  show this help message and exit
```

0.57 添加参数选项

使用 `add_argument` 类来添加参数，以及如何解析参数，语法如下：

```
ArgumentParser.add_argument(name or flags...[, action][, nargs][, const][, default][, type
```

0.57.1 name or flags

name 或者 flags 用来指定参数名称，或者参数列表，其中以-开始的参数为可选参数。例如：

```
import argparse

parser = argparse.ArgumentParser()
parser = argparse.ArgumentParser(description = '合并多个markdown文件并转化为docx文件')
parser.add_argument('echo')
parser.add_argument('-s','--source')

args = parser.parse_args()

print (args.echo)
print (args.source)
```

上述代码增加了 1 个必需参数 echo，和 1 个可选参数 source，测试结果如下：

```
  test.py hello -s sun
hello
sun

  test.py hello --source sun
hello
sun

   test.py --source sun
usage: test.py [-h] [-s SOURCE] echo
test.py: error: the following arguments are required: echo

   test.py hello
hello
None
```

结果显示，如果缺少必填参数，则会报错，而可选参数即可用短参数形式，也可用长参数形式。

0.57.2 help

不论是必选参数还是可选参数，强烈建议使用 help 参数添加说明文字，该说明文字会自动生成在 help 结果中。

```
import argparse

parser = argparse.ArgumentParser()
parser = argparse.ArgumentParser(description = '合并多个markdown文件并转化为docx文件')
parser.add_argument('source',help='待转换的文件')
parser.add_argument('-st','--sourcetype',help='转换前的格式')

args = parser.parse_args()

print (args.source)
print (args.sourcetype)
```

当使用-h 或者–help 输出帮助信息时，结果如下：

```
    test.py -h
usage: test.py [-h] [-st SOURCETYPE] source

合并多个markdown文件并转化为docx文件

positional arguments:
  source                待转换的文件

optional arguments:
  -h, --help            show this help message and exit
  -st SOURCETYPE, --sourcetype SOURCETYPE
                        转换前的格式
```

可见，argparse 模块已经非常贴心地按照 help 参数值，生成了帮助信息。

0.57.3 default 和 type

default 参数用来指定参数默认值，type 用来指定参数类型（默认值是 string），这两个参数经常一起使用，用来限定参数值。

```
import argparse

parser = argparse.ArgumentParser()
```

```
parser = argparse.ArgumentParser(description = '合并多个markdown文件并转化为docx文件')
parser.add_argument('--source','-s',help='待转换的文件',default='source.md')
parser.add_argument('-st','--sourcetype',help='转换前的格式')
parser.add_argument('-l','--level',help='压缩级别',type=int,default=1)

args = parser.parse_args()

print (args.source)
print (args.sourcetype)
print (args.level)
```

在上述代码中，增加了三个可选参数，并设定了默认值和类型，结果输出如下：

```
    test.py
source.md
None
1

    test.py -l 3
source.md
None
3
```

可以看到，指定的默认值都起了作用。

0.58 参数解析

只有使用 parse_args() 方法对添加的参数进行解析后，才能在命令行中使用参数，用法很简单，已在前面的代码中多次出现。

0.59 小结

agrparse 模块的功能还有很多，这里只是介绍了入门的用法，还有很多细节没有提到，详细信息请查看官方文档。

0.60 参考文献

1. argparse 模块官方手册[7]

[7] https://docs.python.org/3/library/argparse.html

2. argparse 用法总结[8]

[8] https://www.jianshu.com/p/fef2d215b91d

第三部分

数据抓取

网络爬虫

网络爬虫就是能够按照一定规则，自动收集网络中的数据的程序。

0.61 网络爬虫原理

1. 请求网页，获取网页源代码
2. 提取信息
3. 存储数据
4. 自动化程序

网页本质上是一个存储在指定位置的文本文件（不一定是静态的，可能是由远程计算机根据一定条件计算出来的），因此，网络爬虫的任务，就是获取远程文本文件，然后，对文件进行分析，提取出我们想要的数据。

0.62 HTTP 请求原理

HTTP 遵循请求 (Request)/应答 (Response) 模型。

Web 浏览器向 Web 服务器发送请求，Web 服务器处理请求并返回适当的应答。所有 HTTP 连接都被构造成一套请求和应答。

图 2: PNG

浏览器作为一个客户端，向服务器端发送了一次浏览该地址所对应的网页的请求

服务器同意了客户端的请求

客户端把服务器端的文件下载到本地

浏览器对文件进行解释、展现

0.62.1 URL

统一资源定位符 (Uniform Resource Locator) 是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的 URL。基本 URL 包含模式（或称协议）、服务器名称（或 IP 地址）、路径和文件名，

如：协议://授权/路径? 查询

```
http://alumni.xjtu.edu.cn:9090/donation/namelist?pageNo=1&pageSize=10&billnum=&donateUserName=&
```

爬虫最主要的处理对象就是 URL，它根据 URL 地址取得所需要的文件内容，然后对它进行进一步的处理。

0.62.2 请求方式

0.62.2.1 GET

GET 方法是默认的 HTTP 请求方法，我们日常用 GET 方法来提交表单数据，然而用 GET 方法提交的表单数据只经过了简单的编码，同时它将作为 URL 的一部分向 Web 服务器发送。

0.62.2.2 POST

POST 方法是 GET 方法的一个替代方法，它主要是向 Web 服务器提交表单数据，尤其是大批量的数据。POST 方法克服了 GET 方法的一些缺点。通过 POST 方法提交表单数据时，数据不是作为 URL 请求的一部分而是作为标准数据传送给 Web 服务器，这就克服了 GET 方法中的信息无法保密和数据量太小的缺点。

0.62.3 requests 包

Requests 是 Python 中的 HTTP 库，Requests 库允许你发送符合标准的 HTTP/1.1 请求，无需手工劳动。你不需要手动为 URL 添加查询字串，也不需要对 POST 数据进行表单编码。Keep-alive 和 HTTP 连接池的功能是 100% 自动化的。

官方网站：http://cn.python-requests.org/zh_CN/latest/

安装方式：

```
pip install requests
```

0.63 编码

0.63.1 编码方式

1. ASCII 编码：是对英语字符和二进制之间的关系做的统一规定
2. GBK 编码：是汉字编码标准之一，是在 GB2312-80 标准基础上的内码扩展规范，使用了双字节编码
3. GB2312 编码：适用于汉字处理、汉字通信等系统之间的信息交换
4. GB18030 编码：国家标准 GB18030-2005《信息技术中文编码字符集》是我国继 GB2312-1980 和 GB13000.1-1993 之后最重要的汉字编码标准，是我国计算机系统必须遵循的基础性标准之一。GB18030 有两个版本：GB18030-2000 和 GB18030-2005。GB18030-2000 是 GBK 的取代版本，它的主要特点是在 GBK 基础上增加了 CJK 统一汉字扩充 A 的汉字。GB18030-2005 的主要特点是在 GB18030-2000 基础上增加了 CJK 统一汉字扩充 B 的汉字。
5. UTF-8 编码：是 Unicode Transformation Format - 8 bit 的缩写。它是可变长的编码方式，可以使用 1~4 个字节表示一个字符，可根据不同的符号而变化字节长度
6. Unicode 编码：这是一种世界上所有字符的编码。当然了它没有规定的存储方式

0.63.2 编码转换

通常是要以 Unicode 作为中间编码进行转换，即先将其他编码的字符串解码（decode）成 Unicode，再从 Unicode 编码（encode）成另一种编码。

0.64 存储

存储的时候，可以将字符串编码之后，再存储到文件。

0.64.1 存储到文件

使用 open 函数，可以创建文件，并将内容写入到文件中。

```
f = open('xjtu.html', 'w')
f.write(content.encode('utf-8'))
f.close()
```

HTTP 库

Python 提供了功能齐全的 HTTP 库，使用这些库，我们只需要关心请求的地址是什么，参数是什么，不用关心更底层的技术，大大降低了信息抓取的难度。最基础的 HTTP 库有 urllib、request 等等。

0.65 urllib

Urllib 是 Python3 内置的 http 库，它包含四个模块：

1. request
2. error
3. parse
4. robotparser

0.65.1 发送请求

使用 urllib 的 request 模块，可以方便地实现请求的发送，并得到响应。

```
import urllib.request

rqs = urllib.request.urlopen('http://www.baidu.com')
html = rqs.read()
print(html)
```

`request()` 方法返回 `HTTPResponse` 类型的对象，具有一些处理信息的属性和方法。其中最基本的是 `urlopen()` 方法，可以完成最基本的请求。它的使用方法如下：

```
urllib.request.urlopen(url, data=None, [timeout, ]*, cafile=None, capath=None, cadefault=F
```

0.65.1.1 data 参数

如果要添加 `data` 参数，则需要使用 `byte()` 方法构造一个字节流编码格式的内容。使用 `data` 参数后，请求方式就变成了 `POST` 方式。

```
import urllib.request
import urllib.parse

data = bytes(urllib.parse.urlencode({'word':'hello'}),encoding='utf8')
rqs = urllib.request.urlopen('http://httpbin.org/post',data = data)

print(rqs.read())
```

0.65.1.2 timeout 参数

timeout 参数用于设置超时时间，单位为秒，默认为全局默认时间。

```
import urllib.request
import urllib.parse
import socket
import urllib.error

data = bytes(urllib.parse.urlencode({'word':'hello'}),encoding='utf8')

try:
    rqs = urllib.request.urlopen('http://httpbin.org/post',data = data,timeout=0.1)
except urllib.error.URLError as e:
    if isinstance(e.reason,socket.timeout):
        print('time out')
```

通过设置 timeout 参数来实现超时处理，是非常有用的策略。

0.65.1.3 Request 方法

如果要在请求中加入头信息，就需要使用 Request 类。

```
import urllib.request
import urllib.parse
import socket
import urllib.error

url = 'http://httpbin.org/post'
headers = {
    'User-Agent':'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)',
    'Host':'httpbin.org'
```

```
}
dict = {
    'name':'Yang'
}

data = bytes(urllib.parse.urlencode(dict),encoding='utf8')
req = urllib.request.Request(url=url,data = data,headers=headers)

try:
   response = urllib.request.urlopen(req,timeout=3)
except urllib.error.URLError as e:
    if isinstance(e.reason,socket.timeout):
        print('time out')

print(response.read())
```

依然用 urlopen() 方法来发送这个请求，只不过参数不是 url，而是 Request 类型的对象。Request 对象的参数如下：

```
class urllib.request.Request(url, data=None, headers={}, origin_req_host=None, unverifiabl
```

1. url 是必选参数
2. data 必须是字节流类型。
3. headers 是一个字典
4. origin_req_host 是请求方的 host 名称或者 ip 地址
5. unverfiable 表示请求是否无法验证（如读取图像的权限）
6. method 的值为 GET、POST、PUT 等等。

0.65.2 处理异常

urllib 的 error 模块定义了由 request 模块阐释的异常。如果出现了问题，request 模块便会抛出 error 模块定义的异常。

0.65.2.1 URLError

URLError 类继承自 OSError，由 request 模块产生的异常都可以通过这个类捕获。它具有一个属性 reason，即返回错误的原因。例如，我们访问一个不存在的页面：

```
from urllib import request,error

try:
```

```
    response = request.urlopen('http://yangzh.cn/php.html')
except error.URLError as e:
    print(e.reason)
```

运行结果为 Not Found。程序没有直接报错，而是输出上述内容，这样我们就具备了处理意外的能力，使得程序更加健壮。

0.65.2.2 HTTPError

HTTPError 是 URLError 的子类，所以可以先捕获 HTTPError，然后再捕获 URLError，如果正常，则用 else 来处理，这是一种更加稳妥的异常处理方式。例如：

```
from urllib import request,error

try:
    response = request.urlopen('http://yangzh.cn/php.html')
except error.HTTPError as e:
    print(e.reason,e.code,e.headers,sep='\n')
except error.URLError as e:
    print(e.reason)
else:
    print('请求成功')
```

HTTPError 有 3 个属性：

1. code 为 HTTP 状态码。
2. reason 为错误原因，其值可能是字符串，也可能是对象。
3. headers 为响应头信息。

0.65.3 解析链接

urllib 中的 parse 模块，可以实现 URL 的抽取、合并以及链接转换，在数据抓取中使用频率很高。

0.65.3.1 urlparse()

urlparse() 可以实现 URL 的识别和分段，例如：

```
from urllib.parse import urlparse

url = 'https://cn.bing.com/search.php?q=python#id-1'
```

```
result = urlparse(url)
print(type(result),result,sep='\n')
```

运行结果如下：

```
<class 'urllib.parse.ParseResult'>
ParseResult(scheme='https', netloc='cn.bing.com', path='/search.php', params='', query='q=
```

从结果可以看出，该方法将 URL 分解为六个部分：

1. scheme 表示协议
2. netloc 表示域名、主机
3. path 表示主机中的路径
4. params 表示参数
5. query 表示查询条件，GET 方式提交请求会产生此类信息
6. fragment 表示地址片段，指向页面内部锚点

0.65.3.2 urlencode()

urlencode() 用于构造 GET 请求，如：

```
from urllib.parse import urlparse,urlencode

params ={
    'name':'yangzh',
    'age':'21'
}

base_url = 'http://www.yangzh.cn?'
url = base_url + urlencode(params)

print(url)
```

结果输出为 `http://www.yangzh.cn?name=yangzh&age=21`。

0.65.3.3 parse_qs()

该函数可以将 GET 请求参数转化为字典，例如：

```
from urllib.parse import urlparse,urlencode,parse_qs
```

```
params ={
    'name':'yangzh',
    'age':'21'
}

base_url = 'http://www.yangzh.cn?'
url = base_url + urlencode(params)

result = urlparse(url)

print(url)
print(result.query,parse_qs(result.query),sep='\n')
```

将打印出：

```
http://www.yangzh.cn?name=yangzh&age=21
name=yangzh&age=21
{'name': ['yangzh'], 'age': ['21']}
```

0.65.3.4 URL 编码

quote() 方法可以将特殊内容（如空格、中文等）转化为 URL 编码的格式，unquote() 方法可以对 URL 编码进行解码。

0.66 使用 requests

与 urllib 库相比，requests 库要更为简洁和人性化，功能也更为丰富。

0.66.1 安装 requests

使用 pip 工具，可以非常简单地实现 requests 库的安装：

```
pip install requests
```

0.66.2 发起请求

利用 params 参数，可以非常方便地构造请求地址：

```
import requests
data = {
    "name": "yangjh",
    "age": 20
}
r = requests.get('http://httpbin.org/get', params=data)
print(r.text)
```

上述代码将会构造 http://httpbin.org/get?name=yangjh&age=20 的地址，并以字符串的形式返回请求结果。对于返回结果是 JSON 格式，还可以使用 json() 方法转化为字典。

利用 headers 参数，可以构造出浏览器标识信息。例如：

```
headers = {
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:63.0) Gecko/20100101 F
}
r = requests.get('https://www.zhihu.com/explore', headers=headers)
```

除了 GET 方法之外，还可以使用 POST、PUT、DELETE 等方法。

0.66.2.1 处理响应

可以通过 text、content、status_code、headers、cookies、url、history 等属性获得服务器返回请求的内容。其中 status_code 的值和 HTTP 状态码一一对应，比如 404 可以用 requests.codes.not_found 来对比。

0.66.2.2 会话维持

使用 Session 方法，可以维持同一个会话，不用每次都设置 cookies 的值，相当于在浏览器中新建选项卡打开网址，而不是重新打开一个浏览器。

```
s = requests.Session()
s.get('http://httpbin.org/cookies/set/age/20')
r = s.get('http://httpbin.org/cookies')
print(r.text)
```

使用 Session，可以模拟登陆成功之后再进行下一步操作。

0.66.2.3 超时设置

为了防止长时间等待，提高效率，我们有必要通过 timeout 属性设置超时时间，例如：

```
r = requests.get('https://www.baidu.com', timeout=1)
print(r.status_code)

r = requests.get('https://www.taobao.com', timeout=(5, 30))
print(r.status_code)
```

如果 timeout 只有 1 个值，则指的是从发出请求到服务器返回响应的总时间；如果 timeout 的值是有俩值的元组，则第一个元素为请求时间，第二个元素为响应时间。

0.66.2.4 Request 对象

可以先构造好请求，再使用 send 方法发送请求，这样可以达到处理不同请求的目的。

```
url = 'http://httpbin.org/post'
data = {
    'age': 20
}
headers = {
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:63.0) Gecko/20100101 Firefo
}
s = requests.Session()
req = requests.Request('POST', url, data=data, headers=headers)
preped = s.prepare_request(req)
r = s.send(preped)
print(r.text)
```

使用 Request 这个对象，我们可以方便地构造出请求。

0.67 网页编码检测及转换

有时，页面虽然申明的编码方式是 `utf-8`，但输出的时候却产生乱码，因此需要先检测页面编码方式，再将其转化为 utf 编码。例如：

```
html = requests.get(url, headers=headers, cookies=jar)
print(html.text.encode(html.encoding).decode('utf8'))
```

使用 requests 库中的 `encoding` 属性，可以获得返回对象的编码方式，然后按照其编码方式编码，再按照 utf8 方式解码，即可得到 UTF 编码方式的内容。

0.68 参考资料

1. https://docs.python.org/3/library/urllib.html
2. http://cn.python-requests.org/zh_CN/latest/

Pyquery

Pyquery 是一个使用 jQuery 方式进行高效解析的库。与使用正则表达式相比，借助于诸如 pyquery 之类的解析库，开发人员可以写出可读性更强的网页解析代码。

0.69 安装

```
pip3 install pyquery
```

0.70 初始化

所谓初始化，就是引入 pyquery 库，获取解析文本。例如：

```
from pyquery import PyQuery as pq
import requests

html = requests.get('http://www.yangzh.cn').text
doc = pq(html)
print(doc('title').text())
```

上面的代码使用 requests 库获得网页内容，再使用 pyquery 库进行解析，获取网页标题元素的内容。

0.71 获取信息

使用 pyquery 获取网页信息的方式和 jQuery 是一致的，先通过选择符进行对象的选择，再对选择的对象进行操作。

0.71.1 通过选择符选定元素

```
titles = doc('ul.kanban-List li h3').items()
times = doc('ul.kanban-List li p span.redTxt').items()
locations = doc('ul.kanban-List li b.localIcon').parent().items()
```

0.71.2 通过迭代获取最终结果

使用 text() 方法，可以获取元素的内容，当然，还可以使用 attr() 获取指定属性的值。利用 zip 函数，可以对多个迭代器进行遍历：

```
def merge_info():
    for title, time, location in zip(titles, times, locations):
        # print(title.text(), time.text(), location.text())
        if '江安' in location.text():
            yield {
                'title': title.text(),
                'time': time.text(),
                'location': location.text()
            }
```

0.72 参考资料

1. https://pyquery.readthedocs.io/en/latest/
2. https://jquery.com/

数据存储

获取到的信息，我们通常使用文件、数据库的形式存储起来，以便再次利用。

使用文件的方式保存数据，用法比较简单，而数据库的方式则分为使用关系型数据库和非关系型数据库，其各自的代表数据库产品是 MySQL 和 MongoDB，皆为开源产品，从性能和安全性角度都具有很强的竞争力。因而，下面介绍这两个数据库在 Python 中的使用。

0.73 MySQL 的存储

成功从页面获取信息后，我们可以将其保存到文件中，也可以存储到数据库中。显然，使用数据库拥有更多后续分析的便利，我们采用常见的关系型数据库 MySQL 作为信息的保存方式。

0.73.1 安装 PyMySQL 和 MySQL

使用 Homebrew 和 pip 工具分别安装 MySQL 和 PyMySQL。

0.73.2 连接数据库

对数据库的使用和其它语言是一致的，无非就是连接数据库，发送指令，接收结果。

```
import pymysql
db = pymysql.connect(host='localhost', user='user',password='password', port=3306)
```

使用 pymysql 的 `connect()` 方法，设置必要的信息，可以连接到 MySQL 数据库。

还可以直接连接到 MySQL 中已经存在的库：

```
connection = pymysql.connect(host='localhost',
                             user='user',
                             password='passwd',
                             db='db',
                             charset='utf8mb4')
```

0.73.3 对数据库进行操作

连接到数据库后，可以使用游标 cursor() 方法对数据库进行各种操作。

0.73.3.1 创建数据库

使用 SQL 语句，就可以创建数据库，创建数据库的语句如下：

```
import pymysql

db = pymysql.connect(host='localhost', user='root',
                     password='mysql', port=3306)

cursor = db.cursor()
sql = "CREATE DATABASE spiders DEFAULT CHARACTER SET utf8"
cursor.execute(sql)
db.commit()
db.close()
```

使用游标方法 `cursor()` 创建游标对象，用 `execute()` 方法执行 SQL 语句，再使用 `commit()` 提交，最后使用 `close()` 方法关闭数据库连接。

0.73.3.2 创建数据表

使用 SQL 语句可以创建数据表，当然前提是连接到数据库。

```
# coding=utf-8
import pymysql

db = pymysql.connect(host='localhost', user='root',
                     password='mysql', db='spiders')

cursor = db.cursor()
sql = "CREATE TABLE IF NOT EXISTS students (id VARCHAR(255) NOT NULL, name VARCHAR(255) NOT NUL
cursor.execute(sql)
db.commit()

db.close()
```

SQL 语句的语法，可以参照 MySQL 参考手册，也可借助于类似 phpMyadmin 之类的工具进行生成和检测。

0.73.3.3 插入、更新、删除数据

使用 INSERT 语句，可以插入数据到数据表：

```
# coding=utf-8
import pymysql

db = pymysql.connect(host='localhost', user='root',
                     password='mysql', db='spiders')

cursor = db.cursor()

id = '2018'
user = 'yangjh'
age = 18

sql = 'INSERT INTO students(id,name,age) values(%s,%s,%s)'
try:
    cursor.execute(sql, (id, user, age))
    db.commit()
except:
    db.rollback()
db.close()
```

插入、更新和删除操作都是对数据库进行更改的操作，这些操作的写法是：

```
try:
    cursor.execute(sql)
    db.commit()
except:
    db.rollback()
```

使用 rollback() 方法可以回滚执行失败的操作，保持事务的原子性（atomicity），使用异常处理，可以方便地实现业务回滚。

0.73.3.4 查询数据

使用 `while` 循环加 `fetchone()` 方法循环获取数据：

```
sql = 'SELECT * FROM students WHERE age <=20'
try:
    cursor.execute(sql)
```

```
    row = cursor.fetchone()
    while row:
        print(row, row[0])
        row = cursor.fetchone()
except:
    print('出错啦！')
db.close()
```

0.74 MongoDB

MongoDB 是一款开源的非关系数据库管理软件，性能优异，在诸多大型项目中表现良好，是非关系型数据库的首选。

0.74.1 在 macOS 中的安装

MongoDB 只支持 macOS 10.11 及以后的产品。

0.74.1.1 使用包管理工具安装

我们使用在 macOS 中的包管理工具 Homebrew 进行安装。

```
brew install mongodb
```

上述指令将安装 MongoDB 的最新发行版，并会自动安装 MongoDB 所需要的依赖环境。

0.74.1.2 运行前的准备

在启动 MongoDB 之前，需要创建 MongoDB 读写数据的目录，默认情况下，MongoDB 会以`/data/db` 为默认目录。下面的命令将创建默认的`/data/db` 目录：

```
mkdir -p /data/db
```

创建目录后，为给目录指定读写权限。然后使用如下命令将 MongoDB 加入到系统服务中，以便每次启动时都可自动启动 MongoDB 服务。

```
brew services start mongodb
```

0.74.1.3 安装 PyMongo 库

启动虚拟 python 环境，然后通过 pip 安装 PyMongo 库：

```
    source bin/activate
(spider)
    pip3 install pymongo
Collecting pymongo
  Downloading https://files.pythonhosted.org/packages/d7/ac/d2e324c1f9bcf653fa106785371a16
    100% |                  | 317kB 1.6MB/s
Installing collected packages: pymongo
Successfully installed pymongo-3.7.2
```

0.74.1.4 安装 Robo 3T

MongoDB 并没有默认的图形化管理工具，只提供 Mongo shell 的命令行方式。为了方便，我们还可以使用诸如 Robo 3T 这样的图形化管理工具进行查询、修改等管理工作。使用 Homebrew 安装：

```
brew install robo-3t
```

安装后，在应用程序中找到 robo-3t，进行必要的配置（指定主机名和端口），即可对 MongoDB 进行管理。

0.74.2 连接 MongoDB

使用 PyMongo 库，可以在 Python 中对 MongoDB 进行操作，连接到数据库，使用 MongoClient 方法即可：

```
import pymongo

client = pymongo.MongoClient('mongodb://localhost:27017')
```

0.74.3 指定数据库

MongoDB 中建立和使用数据库非常简单，直接指定数据库名称即可，如果该数据库不存在，MongoDB 就会创建该数据库：

```
db = client['weibo']
```

0.74.4 指定集合

MongoDB 中的集合（collection），相当于 MySQL 等关系型数据库中的数据表（table），具体的信息，必须指定一个集合，才能进行存入、修改等操作。使用 Collection 对象，即可指定集合：

```
import pymongo
client = pymongo.MongoClient('mongodb://localhost:27017')
db = client['weibo']
collection = db['yangjh']
```

0.74.5 插入数据

推荐使用 `insert_one()` 和 `insert_many()` 方法来分别插入单条记录和多条记录。

```
card = {
    "id": "20183210",
    "name": 'yangjh',
    "age": 20,
    'gender': 'male'
}

result = collection.insert_one(card)
print(result)
print(result.inserted_id)
```

运行结果为：

```
<pymongo.results.InsertOneResult object at 0x10436da48>
5bf8b3f897c32c4128ae8f6f
```

`insert_one()` 方法返回一个对象，调用其 `inserted_id` 属性，可以获得 `_id` 字段的值。

还可以使用 `insert_many()` 方法，一次插入多条记录：

```
card1 = {
    "id": "20183211",
    "name": 'yangzh',
    "age": 20,
    'gender': 'male'
}
```